Volume 17 Number 2

ACTA CYBERNETICA

Szeged, 2005

ACTA CYBERNETICA

Information for authors. Acta Cybernetica publishes only original papers in the field of Computer Science. Manuscripts must be written in good English. Contributions are accepted for review with the understanding that the same work has not been published elsewhere. Papers previously published in conference proceedings, digests, preprints are eligible for consideration provided that the author informs the Editor at the time of submission and that the papers have undergone substantial revision. If authors have used their own previously published material as a basis for a new submission, they are required to cite the previous work(s) and very clearly indicate how the new submission offers substantively novel or different contributions beyond those of the previously published work(s). Each submission is peer-reviewed by at least two referees. The length of the review process depends on many factors such as the availability of an Editor and the time it takes to locate qualified reviewers. Usually, a review process takes 6 months to be completed. There are no page charges. Fifty reprints are supplied for each article published.

Manuscript Formatting Requirements. All submissions must include a title page with the following elements:

- title of the paper
- author name(s) and affiliation
- name, address and email of the corresponding author
- An abstract clearly stating the nature and significance of the paper. Abstracts must not include mathematical expressions or bibliographic references.

References should appear in a separate bibliography at the end of the paper, with items in alphabetical order referred to by numerals in square brackets. Please prepare your submission as one single PostScript or PDF file including all elements of the manuscript (title page, main text, illustrations, bibliography, etc.). Manuscripts must be submitted by email as a single attachment to either the most competent Editor, the Managing Editor, or the Editor-in-Chief. In addition, your email has to contain the information appearing on the title page as plain ASCII text. When your paper is accepted for publication, you will be asked to send the complete electronic version of your manuscript to the Managing Editor. For technical reasons we can only accept files in LaTeX format.

Subscription Information. Acta Cybernetica is published by the Institute of Informatics, University of Szeged, Hungary. Each volume consists of four issues, two issues are published in a calendar year. Subscription rates for one issue are as follows: 5000 Ft within Hungary, €40 outside Hungary. Special rates for distributors and bulk orders are available upon request from the publisher. Printed issues are delivered by surface mail in Europe, and by air mail to overseas countries. Claims for missing issues are accepted within six months from the publication date. Please address all requests to:

Acta Cybernetica, Institute of Informatics, University of Szeged
P.O. Box 652, H-6701 Szeged, Hungary
Tel: +36 62 546 396, Fax: +36 62 546 397, Email: acta@inf.u-szeged.hu

Web access. The above informations along with the contents of past issues are available at the Acta Cybernetica homepage http://www.inf.u-szeged.hu/actacybernetica/ .

W. Brauer
Institut für Informatik
Technische Universität München
D-80290 München
Germany

L. Budach
University of Postdam
Department of Computer Science
Am Neuen Palais 10
14415 Postdam, Germany

H. Bunke
Universität Bern
Institut für Informatik und
angewandte Mathematik
Längass strasse 51.
CH-3012 Bern, Switzerland

B. Courcelle
Universitè Bordeaux-1
LaBRI, 351 Cours de la Libèration
33405 TALENCE Cedex
France

J. Demetrovics
MTA SZTAKI
Budapest, Lágymányosi u. 11.
H-1111 Hungary

B. Dömölki
IQSOFT
Budapest, Teleki Blanka u. 15-17.
H-1142 Hungary

Z. Ésik
University of Szeged
Department of Foundations of
Computer Science
Szeged, Aradi vértanúk tere 1.
H-6720 Hungary

A. Kelemenová
Institute of Mathematics and
Computer Science
Silesian University at Opava
761 01 Opava, Czech Republic

L. Lovász
Eötvös Loránd University
Department of Computer Science
Budapest, Kecskeméti u. 10-12.
H-1053 Hungary

G. Păun
Institute of Mathematics
Romanian Academy
P.O.Box 1-764, RO-70700
Bucuresti, Romania

A. Prékopa
Eötvös Loránd University
Department of Operations Research
Budapest, Kecskeméti u. 10-12.
H-1053 Hungary

A. Salomaa
University of Turku
Department of Mathematics
SF-20500 Turku 50, Finland

L. Varga
Eötvös Loránd University
Department of General Computer Science
Budapest, Pázmány Péter sétány 1/c.
H-1117 Hungary

H. Vogler
Dresden University of Technology
Department of Computer Science
Foundations of Programming
D-01062 Dresden, Germany

G. Wöginger
Department of Matematics
University of Twente
P.O. Box 217, 7500 AE Enschede
The Netherlands

Preface

The 4th Conference for PhD Students in Computer Science (CSCS) was organized by the Department of Computer Science of the University of Szeged (SZTE) and held in Szeged, Hungary from July 1 to 4, 2004. The members of the Scientific Committee were the following representants of the Hungarian doctoral schools in computer science: Mátyás Arató (DE), András Benczúr (ELTE), Miklós Bartha (SZTE), Tibor Csendes (SZTE), János Csirik (SZTE), János Demetrovics (SZTAKI), Sarolta Dibuz (Ericsson), József Dombi (SZTE), Zoltán Ésik (SZTE), Ferenc Friedler (VE), Zoltán Fülöp (SZTE), Ferenc Gécseg (chair, SZTE), Tibor Gyimóthy (SZTE), Balázs Imreh (SZTE), János Kormos (DE), László Kozma (ELTE), Attila Kuba (SZTE), Eörs Máté (SZTE), Gyula Pap (DE), András Recski (BMGE), Endre Selényi (BMGE), Katalin Tarnay (NOKIA), György Turán (SZTE), and László Varga (ELTE). The members of the Organizing Committee were Balázs Bánhelyi, Tibor Csendes (chair), Tünde Felföldi, Mariann Kocsorné Sebő, Gábor Sey, and Péter Gábor Szabó.

There were more than 120 participants and 101 talks in several fields of computer science and its applications. Beyond the Hungarian PhD schools in computer science, 6 other European countries were represented. The talks were going in two parallel sections in artificial intelligence, automata and formal languages, computer networks, database theory, discrete mathematics, fuzzy decision support systems, information systems, optimization, picture processing, and software engineering. The talks of the students were completed by 4 plenary talks of leading scientists.

Three scientific journals, viz. Periodica Polytechnica (Budapest), Publicationes Mathematicae (Debrecen) and Acta Cybernetica (Szeged) offered students to publish the paper version of their presentations after a selection and review process. Altogether 41 manuscripts were submitted for publication. The present special issue of Acta Cybernetica contains 14 such papers.

The full program of the conference, the collection of the abstracts and further information can be found at `http://www.inf.u-szeged.hu/~cscs`.

On the basis of our repeated positive experiences, the conference will be organized in the future, too, hopefully with more foreign participants. According to the present plans, the next meeting will be held in July 2006 in Szeged.

Tibor Csendes and Zoltán Fülöp

Acta Cybernetica 17 (2005) 185–198.

Measurement and Optimization of Access Control Lists

Sándor Palugyai*, Máté J. Csorba*, Sarolta Dibuz* and Gyula Csopaki†

Abstract

This paper deals with the examination of Access Control Lists (ACLs) that are used in IP routers mainly for providing network admission control and maintaining a certain level of quality of service. In our work we present a method for measuring the performance impact of ACLs on the packet forwarding capabilities of a router. Besides, our study proposes new methods to model and optimize the operation and reduce the redundancy of ACLs.

1 Introduction

Nowadays the Internet usage is progressing at a great pace. More and more people become potential users and require faster connections. Recently, security has also become an important issue in business networks and at home as well. Because of these facts devices have to be designed and created, which allow us to build and maintain a more secure network. Their operation has to be optimized also. In this work a method is proposed for the optimization of Access Control Lists (ACLs) used in routers, which can maintain the operation of large networks. Besides, we present the model designed for the optimization process.

In the next section we describe how ACLs work and can be used in IP networks. We also give a short example on their usage. In Section 3 we outline our method for the measurement of ACL performance that was implemented in TTCN-3, and we present our measurement results with a conventional access router. In Section 4 we introduce a directed graph representation of ACLs and we give an algorithm based on the model for optimization of packet forwarding performance. In Section 5 experimental measurement results are presented to demonstrate the applicability of our method. Finally, in Section 6 conclusion is given together with the possible future developments.

*Ericsson Hungary Ltd., Test Competence Center, H-1117 Budapest, Irinyi J. u. 4-20. Email: {sandor.palugyai, mate.csorba, sarolta.dibuz}@ericsson.com

†Technical University of Budapest, Department of Telematics and Media Informatics Email: csopaki@tmit.bme.hu

2 The Application of Access Control Lists

Nowadays, TCP/IP is the most widely used networking protocol, so it is an important security issue to control or restrict TCP/IP access. To achieve the needed control over IP traffic and to prohibit unauthorized access, ACLs are a commonly used solution in firewall routers, border routers and in any intermediate router that needs to filter traffic.

ACLs are basically criteria put into a set of sequential conditions. Each line of such a list can permit or deny specific IP addresses or upper-layer protocols. Incoming or outgoing traffic flows can be classified and managed by a router using ACLs. There are two basic types of ACLs: standard and extended.

With standard IP access lists, a router is capable of filtering the traffic based on source addresses only. Extended access lists, on the other hand, offer more sophisticated methods for access control by allowing filtering based not only on source addresses, but also on destination addresses and other protocol properties. Hence the command syntax of an extended ACL can be far more complex than a standard one.

Standard ACL syntax:

```
Router1(config)# access-list acl-number {deny | permit} [host]
source-address [source-mask] [log]
```

Extended ACL syntax:

```
Router1(config)# access-list acl-number {deny | permit} protocol
[host] source-address [source-mask] [host] destination-address
[destination-mask] [precedence precedence-id] [tos tos-id]
[established] [log] [time-range tr-name]
```

Figure 1: **Standard and extended access list syntax**

In many cases ACLs are used for allocating resources needed by a user at a given time of a day, or to automatically reroute traffic according to the varying access rates provided by the ISPs. Service Level Agreements (SLAs), negotiated in advance, can be satisfied as well if time ranges are also specified in an access list. However time-based ACLs are not taken into consideration in this paper.

ACLs can be applied on one or more interfaces of the router and in both directions, but they work differently depending on which direction they are applied. When applied on outgoing interfaces, every received packet must be processed and switched by the router to the proper outgoing interface before checking against the appropriate list. And in case the rules defined in the list drop the packet, this results in a waste of processing power.

When the administrator defines the access lists needed, they must be applied on the proper interface by issuing the ip access-group command.

An application area for access lists is called session filtering. The main purpose of session filtering is to prevent (possibly malicious users on) outside hosts con-

necting to hosts inside, while still allowing users inside the protected network to establish connections to the outside world.

For the sake of clarity, consider the following example (Fig. 2). The administrator who is managing a local network wants to allow users of the corporate network to access the local web-server, but at the same time access to the local workstations must be prohibited. Besides, the workstations should be able to establish connections destined to the corporate network.

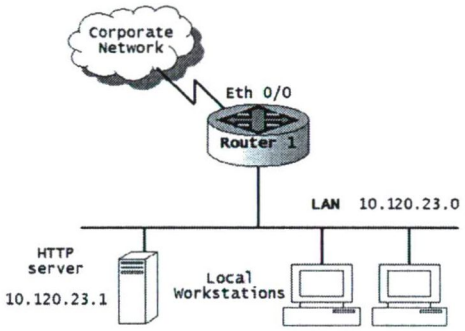


Figure 2: **Example network**

The solution to the problem introduced above is realized by the ACL numbered 111, which contains two separate lines. The first permits TCP traffic originated from any host, destined to the single host 10.120.23.1, which is an HTTP server. The destination TCP port is also restricted to 80, on which the HTTP server software is listening. The second line prohibits connections initiated by any host on the Corporate Network destined to the local network 10.120.23.0. Although it seems that this statement cuts the whole internal LAN from the outside world, the HTTP server is still available to connections because every incoming packet is checked against the statements sequentially.

Example ACL (two lines):

```
Router1(config)# access-list 111 permit tcp any host 10.120.23.1
eq 80
Router1(config)# access-list 111 deny any 10.120.23.0
0.0.0.255
```

Figure 3: **Setting up an ACL (an example)**

So, if the incoming packet belongs to a connection destined to the HTTP server, it matches the first line of the ACL and it is routed and transmitted to its destination. Any other packets that do not match the first line are checked against the second line and are discarded. In fact, every access list has a virtual line at the end that is called the implicit deny rule. The implicit deny discards every packet originated from any address destined to any other address. So, if the examined

packet does not match any of the rules, at the end it matches the implicit deny rule and it is discarded. As a matter of fact the second line is not necessary. Finally, when the proper access list is constructed, it needs to be bound to an interface of the router (Fig. 4).

Applying the ACL on interface Ethernet0/0:

```
Router1(config-if)# ip access-group 111 in
```

Figure 4: **Setting up an ACL (continued)**

Although conventional ACLs are relatively static, dynamic access lists exist to allow the rules to be changed for a short period of time, but require additional authentication processes. In this case exceptions are granted for the user (possibly with a higher privilege-level) to access additional network elements. The current work does not consider these types of ACLs [1].

When an ACL is applied on a router's interface, the router is forced to check every packet sent or received on that interface depending on the type of the ACL (in or out). This can seriously affect the packet forwarding performance. A very simple solution to cope with the performance impact of ACLs is to use the null0 interface, which is implemented software-only and acts as a garbage bin or a virtual interface for the unwanted traffic.

The null0 interface can be used if and only if all of the traffic destined to a particular host or network destination needs to be restricted. In this case, a static route to the null0 interface can be added to the route table. This way the router forwards the unwanted traffic to the virtual garbage bin simply via a routing table entry without checking the packets against the ACL [2].

3 Measurement of ACL Performance

Once an ACL is bound to one of the router's interfaces it may have a serious effect on packet forwarding. To determine the properties of this effect and to be able to compare performance of routers from different vendors we developed a method for measuring the ACL performance.

The performance measurements were implemented in TTCN-3 (Testing and Test Control Notation version 3) language [3], which is originally used for conformance testing purposes and is very similar to the conventional C language. Accordingly, it is well equipped with constructs, supporting message-based communication with the particular implementation under test.

Basically, a TTCN-3 test program has the following necessary modules. A module containing the definition of packet and message types used during testing; another module contains the so-called templates, which are basically constraints to the incoming and outgoing communication; a main module contains the functions and test cases used during testing, and reads the configuration files for parameters that are alternating between each test execution.

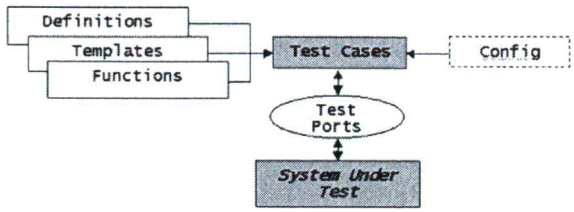


Figure 5: **Simplified diagram of the most important modules of a TTCN-3 program**

A very important part of the test system is the test port, which establishes connection to the operating system allowing the test program to establish communication with the implementation under test.

In traditional conformance testing methodology test ports are used simply as communication bridges without any further intelligence. Our measurement method utilizes modified IP test ports, which allow the use of precise timings and the transmission of IPv4 packets. But, beyond the original capabilities, the modified test port can cope with delay and round-trip-time measurements by using time stamps for any packet passing through.

The measurement method uses different traffic patterns to estimate the delay as a function of ACL size. In all cases artificial flows are generated using TTCN-3. One option is to apply a rough estimation and simulate the distribution of packets, according to a macroscopic view of real Internet traffic. The packet distribution is composed based on the data collected by the NLANR project [4]. During this project 342 million packets were observed and analyzed. The average packet size was 402.7 bytes.

The traffic according to this model consists of the following three main packet types:

— 40 bytes: TCP packets without payload (20 bytes IP header + 20 bytes TCP header). These packets can be observed typically at initiation of a TCP connection. Approximately 35% of the packets can be classified into this type, but because these packets are very small this type gives only 3.5% of the traffic.

— 576 bytes: TCP packets of obsolete implementations, which still use the MSS (Maximum Segment Size) value. 11.5% of the packets are this type though giving 16.5% of the traffic.

— 1500 bytes: packet size according to the Ethernet MTU (Maximum Transfer Unit). Most of the data flowing through the Internet consists of full sized Ethernet frames, though giving 10% of the packets and 37% of the overall traffic.

Considering packets with occurrence rate over 0.5%, the following packet sizes may occur (in order of frequency): 52, 1420, 444, 48, 60, 628, 552, 64, 56 and 1408

bytes. During the NLANR project 1.2% of the packets were smaller than 40 bytes. Although these packets are very small (only 0.1% of the traffic) the routers have to forward them also, and must be capable to handle the serious overhead caused by them.

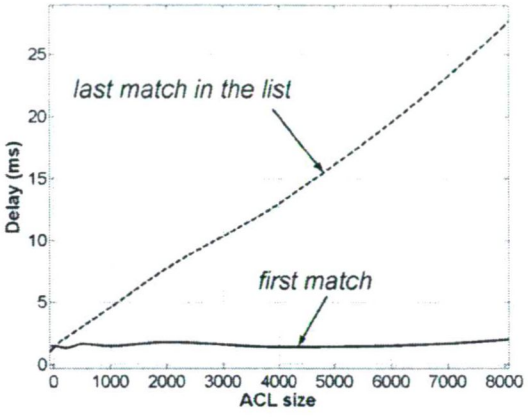


Figure 6: **Delay as a function of ACL size**

In the example (Fig. 6), 64 byte long UDP packets are generated every millisecond. This speed is relatively slow compared to the raw throughput capability of the router under test (Cisco 2600 series access router, with an approximate transmission capability of 15000 packets per second [5]), to avoid undesirable latency or packet loss. The results confirm the conjecture that the delay is increasing significantly with the increasing number of access list entries.

When examining ACL behavior, content carried in packet header fields carries relevant information. Accordingly measurement traffic is composed without respect to protocol payload, while the header fields are variable. Optimization is made based upon the match probabilities that the optimization algorithm reads as input for the process. By means of different match probabilities different traffic mixtures can be represented.

We have also measured the delay with test rules mapped to the routing table using the *null0* interface. In this case the average delay was only 0.32 ms with a variance of 0.1 us. However in this measurement, we had to constrain the rules to use destination address prefixes only, because of the limited capabilities of this filtering solution.

4 Optimization of ACL Performance

The first objective during our research was to find a suitable representation format for the access lists for further examination. Hazelhurst [6] proposed the usage of binary decision diagrams, first introduced by Bryant [7] to represent access lists

Table 1: **The example ACL in the original order with match numbers**

Node	rule	prefix/length	# of matches
C	deny	10.120.238.130/32	4299
O	deny	10.120.238.7/32	357
G	deny	10.120.240.0/24	2500
A	deny	10.120.238.0/28	1214
B	permit	10.120.238.0/26	4910
D	permit	10.120.238.128/26	1703
J	deny	10.120.0.0/16	2028
K	permit	10.121.130.0/24	125
L	deny	10.121.0.0/16	1380
I	permit	10.120.239.132/32	417
N	deny	10.120.239.128/26	1612
H	deny	10.120.239.0/24	3301
E	deny	10.120.238.0/24	3
M	permit	10.120.238.64/26	0
F	permit	10.120.0.0/16	3405

systematically. We decided to use directed graphs to describe the dependencies between the list entries.

The example graph is constructed considering the following parameters: type of rule (permit/deny), network prefix, network mask length and the number of times the rule matched. Every node represents one line of the access list. In this quite simple example we assume that filtering is based on destination addresses only, and more importantly there is no rule querying any upper-layer protocol information such as TCP and UDP port numbers or protocol IDs.

From this representation of the access list, the following graph can be constructed identifying which network prefix is more general and contains the other prefix as well (Fig. 7). The nodes are labeled by the capitals A..O each one representing a single rule (one line of the list).

At first, existing redundancy can be decreased in the list, by eliminating nodes, which depend from a node with a wider prefix having the same rule, and the wider rule is not represented by a star-like node. For example, a node can be deleted if its rule is preceding the other rule in the original list and its rule is completely a subset of the other rule at the same time. In this case, the weight of the actual rule (number of matches) is added to the weight of the more general rule. Secondly, suspicious nodes with 0 match can be checked and eliminated if needed. For example, node M permits traffic to 10.120.238.64/26 but network 10.120.238.0/24 is prohibited by node E, which comes before M in the original list, so M can be spared.

We optimize an extended ACL, so each rule may contain port, protocol and address related constraints. The graph representing an extended ACL may contain cycles, as for example in the following basic rule-set.

Table 2: **Cycle in the ACL**

No.	rule	Source address	Destination address	Protocol
1	deny	10.120.0.0/24	any	any
2	deny	any	10.200.20.0/24	any
3	deny	any	any	TCP

In this example every list entry is a deny, but none of them can be deleted, because the rules are not a complete subset of each other. The constructed graph will contain the entries the following way (Fig. 8).

Since the rules are examined in a sequential order, obviously the order in which they are specified has a semantic meaning [8]. Accordingly, during the optimization process the edges in the composed graph are directed based on these dependencies. In the example (Fig. 7), we consider the delay that a packet suffers equal for every list entry check operation, because the rules are simple and very similar to each other. But, generally the delay caused by a rule in a list is varying. However, the overall delay of the traffic can be estimated only if we also consider the arrival intensity of the traffic and the queuing at the in/out interfaces [9]. Hence, we estimate the delay of the traffic in this example with a ratio, which is equivalent to the case when the traffic is slow enough that no queuing is present at the router's interfaces. In turn, we are able to compare the improvement we can gain by reordering the list entries.

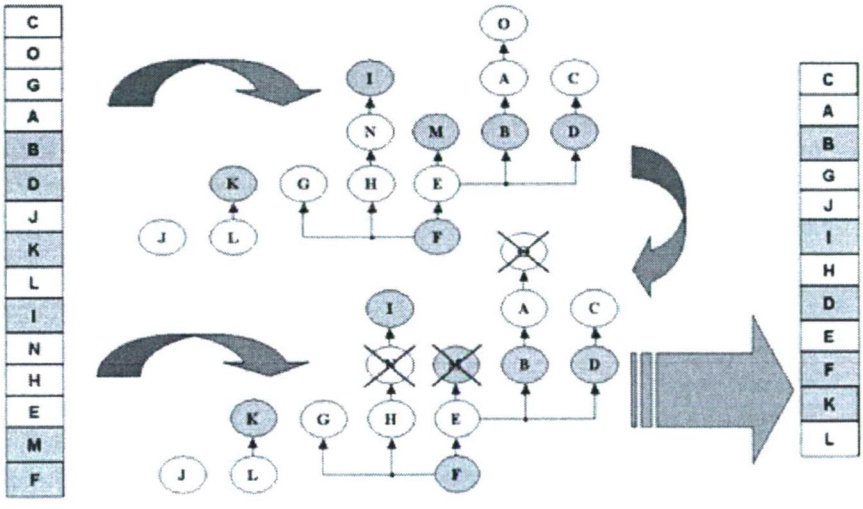


Figure 7: **The graph representing the example and the optimization process**

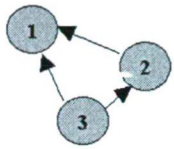


Figure 8: **Cycles in the constructed graph**

We define the following parameters:

L	:	the set of actual list entries;
T	:	the traffic that is matched against list L;
$n(L)$	:	the number of list entries in L;
$r_i(L)$	:	the delay of rule number i. in list L;
$m_i(L,T)$	:	the probability that a packet in T matches rule number i. in L.

Considering these parameters the total delay a packet suffers that matches rule number i. in list L can be estimated as:

$$d_i(L) = \sum_{k=1}^{i} r_k(L) \tag{1}$$

Furthermore, the total delay traffic T suffers while filtered through list L if there is no queuing present at the network interfaces:

$$Delay(L,T) = \sum_{i=1}^{n(L)} m_i(L,T) \cdot d_i(L) = \sum_{i=1}^{n(L)} m_i(L,T) \cdot \sum_{k=1}^{i} r_k(L) \tag{2}$$

According to (2) the example input ACL (in Table 1 and Fig. 7) has a delay value of 192381, while the resulting ACL (in Fig. 7) has a value of 144840. These values do not have a unit, since they represent a ratio only for comparison. According to them the delay has been reduced by approximately 25%. The following algorithms use these formulas for comparing runtime results.

At first, we have developed a brute force algorithm to optimize the graph representing the ACL. This algorithm is executed after the graph has been built up in the memory and existing redundancy is removed. The algorithm evaluates every possible layout of the graph depending on the meanings of the rules and preserving the order of nodes that are dependent on each other. Afterwards, the theoretic delay of every layout is calculated using the weights of the nodes. At the end, the layout with the lowest calculated delay is chosen as the best solution. The reordered graph is then transformed back to a sequential list and can be uploaded to the router.

However, the applicability of this algorithm is highly limited because of the time it takes to evaluate every possible set-up. For example, to check 14 nodes lasts 5

sec and for 15 nodes it is already 74 sec, 16 nodes last 1310 sec and for 20 nodes it would take approximately 1763 days.

The graph optimizing processes were implemented in C++, because of the computationally intense calculations. Besides, the on-line communication with the router is implemented in *Perl* language and uses a telnet connection. This way the software can connect to routers from various vendors including Cisco, besides there is no need to implement the optimization inside the router, but it can be performed remotely from the connected network.

In order to overcome the limitations of the brute force algorithm we have to consider a more efficient way of rebuilding the Access Control List. But first, we need to fabricate a criterion for a basic building block of our optimization process, namely to estimate the resource demand of merging two simple sub-graphs of the ACL.

So, let us consider two separate sub-graphs of an ACL, namely list K and L. Let us also assume that the number of list entries is k and l respectively. In this case the two lists can be merged s_{kl} ways:

$$s_{kl} = g_{l+1}^T \cdot A_{l+1}^{k-1} \cdot e_{l+1} \tag{3}$$

Whereas matrices g_{l+1}, e_{l+1} and A_{l+1} are the following:

$$\mathbf{g}_{l+1} = \begin{bmatrix} 1 \\ 2 \\ \vdots \\ l+1 \end{bmatrix} \quad ; \quad e_{l+1} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \quad ; \quad A_{l+1} = \begin{bmatrix} 1 & 1 & \cdots & 1 & 1 \\ 0 & 1 & \cdots & 1 & 1 \\ \vdots & & \ddots & & \vdots \\ 0 & 0 & \cdots & 0 & 1 \end{bmatrix} \quad ;$$

More importantly, separate entries of K and L preserve their order in the resulting list. Using Equation 3 we constructed the following algorithm:

Algorithm 1 (The pseudo-code of the optimization algorithm).

```
1. Establish an authenticated connection towards the router;
2. Query the Access List data;
   {
    2.1. FOREACH list entry
       {
        2.1.1. Store the actual rule and number of matches into the
           memory
       }
   }
3. IF the newly created list is not the same as the one we have stored
   previously
   {
    3.1. Construct a (possibly non-continuous)graph structure according
       to the ACL rules;
```

```
      3.2. Assign weights to every node based on the number of matches on
           the particular list entry and on the delay of the actual rule;
      3.3. Eliminate possible redundant entries in the list
           {
            3.3.1. IF an entry exists in the list that has never been
                   matched, or incidentally the list contains an error, the entry
                   is removed from the list
           }
   }
4. FOREACH node starting from the leaves towards the root of the graph
   {
      4.1. IF the sub-graph starting from the actual node can be reordered
           in reasonable time, according to (3), THEN the actual sub-graph is
           arranged into one branch with the brute-force algorithm. (The
           amount of reasonable time is estimated based on measurements and
           it is heavily hardware dependent, consequently short enough to
           allow us to neglect the time needed for reordering the nodes.)
   }
5. FOREACH node starting from the leaves towards the root of the graph
   {
      5.1. <weight of the actual node>:= <the original number of matches it
           has received> + <the weight of the underlying branches divided by
           the distance from the actual node>
           }
6. FOREACH node of the graph
   {
      6.1. <l_i> := the leaf with the most significant weight;
      6.2. Move <l_i> to the end of the list;
      6.3. Remove the selected leaf from the list (if <l_i> was the last node
           of a branch, zero or more new leaves appear)
   }
7. Replace the current Access Control List in the router with the newly
   created one (this operation needs packet forwarding to be suspended
   for a very short period of time for security reasons).
```

After the execution of this algorithm we compared the results with the results produced by the previous, brute force method. But, since the brute force algorithm can only be executed for a small amount of list entries the comparison is valid only for a few entries. However, we also conducted measurements with the method mentioned in Section 3, and found that with periodic optimization (using the algorithm detailed above) we can decrease the delay resulting from the use of very long ACLs, whilst still keeping the time needed for the execution of our optimization process below a reasonable level.

As ACLs are usually defined once by a network administrator with respect to the given policies in the organization, and might be upgraded several times by hand,

the process lacks any kind of feedback or optimization based on the actual traffic in the network. In contrast, our method monitors list entry hit rates according to the traffic and can modify the list and upload a new one if it shows to be faster. The measurements show that our algorithm can be executed in an insignificant time (not more than 1 second) below 3000 access list entries, which is typically enough for routers used by Internet Service Providers. Moreover the time needed for the optimization can be kept below 70 seconds even for 10000 list entries.

5 Application Results

Initially, we developed four different algorithms called A1-A4. Afterwards, we made a thorough comparison regarding their performance and efficiency and decided to use algorithm A4.

All four of the algorithms perform the following steps. A node is chosen at each step and transferred into the final re-ordered list. Algorithm A1 chooses nodes according to leaf weights. A2 evaluates paths towards a certain leaf while summarizing node weights at the same time. Similarly A3 evaluates paths, but this algorithm decreases node weights along the path according to the place a node has in the path, e.g. the weight of the fourth node in the path is divided by four and the weights are summarized. A4 four is very similar to A3, but this time a node weight is divided by two at the power of node place.

We generated list representations with rules in random initial order for testing the algorithms. Efficiency was calculated by comparing the resulting list of each algorithm to the initial list (4).

$Result_{original}$ = *overall weight of the random generated list*
$Result_{algorithm}$ = *overall weight after optimization*

$$Efficiency = 100 \cdot \left(\frac{Result_{original} - Result_{algorithm}}{Result_{original}} \right) \tag{4}$$

The scripts that generated random lists could generate random and balanced graphs also. Test runs were performed on the generated graphs with the four algorithms automated also by scripts. The averaged results are shown in the following two diagrams. Fig. 9 represents the efficiency of the algorithms as a function of the number of list entries that is the number of nodes in the graph.

The algorithms were also compared to the Brute Force method, in which case every possible layout of the graph is evaluated and the layout with the least weight is chosen.

According to the results, algorithm A4 performs the best, in most cases the same order is chosen as by the Brute Force method. A few deviations do exist, e.g. one pair of rules is different. However, this comparison was made only with short lists, since the Brute Force method has an unacceptable execution time (Fig. 10).

Fig. 10 shows the execution times as a function of list size. Algorithm A4 was chosen, because it is the most effective and it is fast too. Since even in core routers

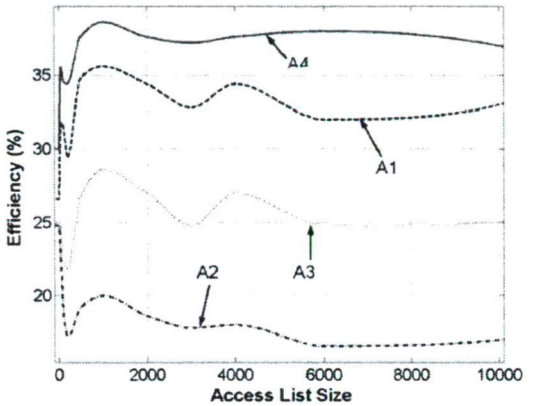


Figure 9: **Efficiency of the new algorithms**

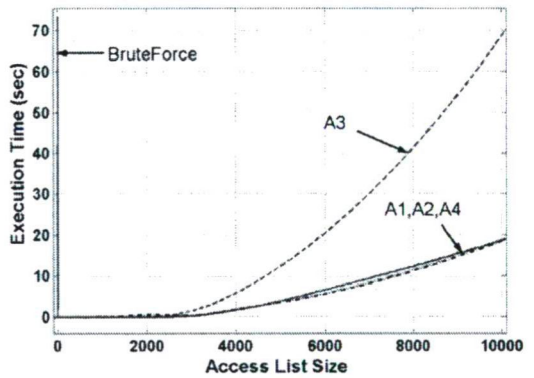


Figure 10: **Execution time of the new algorithms**

more than 10000 entries are quite rare, but few thousands are possible it is notable that algorithm A4 has an execution time of nearly zero till a few thousand entries.

6 Conclusions

Firstly, we have developed a method to measure the performance impact of network management with ACLs. Our measurement method uses regular PCs and it is a software-only solution. Compared to industrial solutions for the same problem, like the RouterTester from Agilent [10], it has similar capabilities combined with relatively cheapness and flexibility of a software solution. Using our test method it is possible to produce several streams to specified destinations, to test the functionalities of access lists. Detailed PDU (Protocol Data Unit) building is available as well. As during an ACL test, raw transmission performance of the tester is not the

most important issue, since the performance impact of ACLs is measured instead of raw throughput capabilities. Our software solution satisfies the requirements of ACL testing. From the measurements, it can be concluded that the number and nature of access list entries have a significant impact on packet transmission in routers, so optimization might be needed.

In the second part of this work we proposed a method to optimize performance of access lists. The method uses directed and weighted graphs to represent ACL rules. We developed an algorithm to optimize the layout of the graph representing the ACL and this way to minimize the latency caused by access lists. Our software is implemented in C++ and Perl.

Our current work focuses on developing new, more efficient and faster algorithms to optimize ACLs, moreover we would like to examine the performance of ACLs using IPv6. Besides, we also would like to develop our method to be able to handle more general scenarios and to build a framework that is capable of examining more general list topologies applied for example in firewall systems or other software architectures as well.

References

[1] Scott Hazelhurst: A proposal for Dynamic Access Lists for TCP/IP Packet Filtering [Sortened Version In Proc. of SAICSIT 2001]

[2] S. Convery: Network Security Architectures [1rst edition, ISBN: 158705115X. Cisco Press (2004)]

[3] ETSI: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; TTCN-3: Core Language [ETSI ES 201 873-1]

[4] NLANR: National Library for Applied Network Research. http://www.nlanr.net/

[5] Cisco: http://www.cisco.com/

[6] Scott Hazelhurst: Algorithms for Analysing Firewall and Router Access Lists [Workshop on Dependable IP Systems & Platforms, In Proc. ICDSN, June 2000]

[7] R Bryant: Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams [ACM Computing Surveys, 24(3) (September 1992)]

[8] Jeff Sedayao: Cisco IOS Access Lists [1rst edition, ISBN: 1-56592-385-5. O'Reilly]

[9] S. Palugyai, M. J. Csorba. Modeling Access Control Lists with Discrete-Time Quasi Birth-Death Processes. Submitted to the The 20th International Symposium on Computer and Information Sciences (ISCIS 2005)

[10] Agilent RouterTester: http://advanced.comms.agilent.com/RouterTester/

Acta Cybernetica 17 (2005) 199–211.

Cycle Structure in Automata and the Holonomy Decomposition

Attila Egri-Nagy* and Chrystopher L. Nehaniv*

Abstract

The algebraic hierarchical decomposition of finite state automata can be applied wherever a finite system should be 'understood' using a hierarchical coordinate system. Here we use the holonomy decomposition for characterizing finite automata using derived hierarchical structure. This leads to a characterization according to the existence of different cycles within an automaton. The investigation shows that the problem of determining holonomy groups can be reduced to the examination of the cycle structure of certain derived automata. The results presented here lead to the improvements of the decomposition algorithms bringing closer the possibility of the application of the cascaded decomposition for real-world problems.

1 Introduction

The aim of this paper is to study the cycle structure in automata associated to the holonomy decomposition in Krohn-Rhodes Theory. With a recent computational tool [5] (developed by the authors) the Krohn-Rhodes theory [9] finally has computational means to foster further research in it and to show its real significance to scientists working outside theoretical computer science. The main aim of this paper is to summarize the theoretical insights gained from the systematic study of finite state automata by examining their derived hierarchical decomposition computed by the implemented holonomy decomposition [6, 4], and show how these insights may be used for improving the algorithms. It also can be considered as a first – although still theoretical – computational application of the Krohn-Rhodes theory remaining within the confines of algebraic automata theory. Further possible applications come up in all different fields where we deal with hierarchical models of systems: physics [13], software-development [10], artificial intelligence [11], evolutionary biology [12], etc.

As the holonomy decomposition mainly deals with certain sets of subsets of an automaton's state set that are permuted by input words, our investigation concentrates on the question of when *nontrivially* permuted sets of appropriate subsets really exist and of recognizing when automata are completely without them.

*School of Computer Science, University of Hertfordshire, College Lane, Hatfield, Herts AL10 9AB, United Kingdom, Email: {A.Nagy | C.L.Nehaniv}@herts.ac.uk

2 Mathematical Preliminaries and Notations

Here we establish the close connection between finite state automata and some algebraic structures called semigroups as it is more convenient to handle automata algebraically. The connection between these structures is outlined here with special emphasis on the cascaded product of automata, together with the notions of division and wreath product. For more details see [4, 1, 6].

2.1 Transformation Semigroups

Semigroups. A *semigroup* is a set S equipped with an associative binary operation $\mu : S \times S \to S$. Instead of $\mu(s_1, s_2)$ we write $s_1 \cdot s_2$ or more briefly $s_1 s_2$. If A and B are subsets of a semigroup, then AB means the set $\{ab : a \in A, b \in B\}$. An element 1 is the identity element of S if $s1 = 1s = s$, for all $s \in S$. The identity is unique if it exists. By S^1 we denote S if it has an identity otherwise $S \cup \{1\}$. By S^I we mean $S \cup \{I\}$ where I acts as an identity on S and itself, the identity of S (if it exists) ceases to be an identity as it fails on I. The *order* of a semigroup S is its cardinality $|S|$. We say that G generates the semigroup $\langle G \rangle = S$ if $G \subseteq S$ and all elements of S can be expressed as a finite product of elements in G. A semigroup S is *aperiodic* if for each element $s \in S$ there is a positive natural number n such that $s^n = s^{n+1}$; for a finite semigroup this means that it contains no nontrivial subgroups.

Homomorphisms. Let S and T be semigroups with operations $\circ, \diamond$ respectively, and having a mapping $\psi : S \to T$ such that $\psi(s_1 \circ s_2) = \psi(s_1) \diamond \psi(s_2)$, for all $s_1, s_2 \in S$. Then we say that ψ is a *homomorphism* from S to T, a mapping which preserves products. If a homomorphism is bijective then it is an *isomorphism.*

Groups. A semigroup is a *monoid* if it has an identity element. A monoid is a *group* if for every $s \in S$ there is an inverse $s^{-1} \in S$ such that $ss^{-1} = s^{-1}s = 1$. A subset T of a semigroup S is a *subsemigroup* if it is closed under the multiplication of S. Subgroups are defined analogously. A subgroup H of a group G is *normal* if $gH = Hg \; \forall g \in G$. A nontrivial group is *simple* if it has no nontrivial normal subgroups.

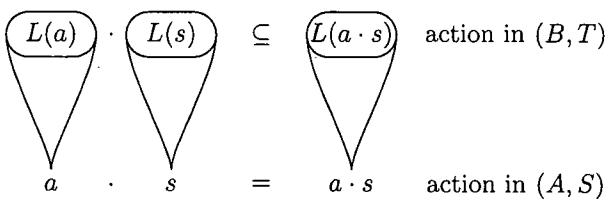
Transformations. For a nonvoid finite set A, a mapping $\varphi : A \to A$ is called a *transformation* of A. If the mapping is bijective, then it is a *permutation.* The *image* of φ is defined as $\{a\varphi : a \in A\}$ denoted by $\mathrm{im}(\varphi)$. If the image of a mapping is a singleton then the mapping is *constant.* The *rank* of a transformation is the cardinality of its image. The set T of all transformations of A form a semigroup under the operation of function composition of transformations and it is called the *full transformation semigroup* denoted by $\mathcal{T}_A = (A, T)$. If S is a subsemigroup of T then (A, S) is called a *transformation semigroup* on A (or briefly a *ts*), and we say that S *acts* on A. (A, S) is a *permutation group* if each elements $s \in S$ acts on A by permutation. We write $a \cdot s$ for the image of state a under the transformation s, and we have $(a \cdot s_1)s_2 = a \cdot (s_1 s_2)$ for all $a \in A, s_1, s_2 \in S$. It is a basic fact of semigroup theory that every finite semigroup can be represented as a ts using

the *right regular representation* (S^1, S) where S acts on S^1 by multiplication on the right [3]. If (A, S) is a transformation semigroup, we denote by $(A, \overline{S})$ the transformation semigroup with transformations $\overline{S} = \{t \mid t \in S \text{ or } t \text{ is constant}\}$.

Division. We say that a transformation semigroup (A, S) *divides* (B, T) denoted by $(A, S) \mid (B, T)$ if we can choose for all $a \in A$ at least one $\tilde{a} \in B$ as a *lift* and and for each $s \in S$ at least one $\tilde{s} \in T$ as a *lift*, such that the following hold:

1. Each member of B (resp. T) is a lift of at most one element of A (resp. S), i.e. the (non-empty) lift sets are non-intersecting,
2. If $\tilde{a}$ is any lift of a and $\tilde{s}$ is any lift of s, then $\tilde{a} \cdot \tilde{s}$ is some lift of $a \cdot s$, i.e. the products are respected.

Denote the set of lifts of a state a by $L(a)$ (and $L(s)$ for a transformation s respectively). Note that in general $L(a) \cdot L(s) \subseteq L(a \cdot s)$, instead of being equal.



Words and the free semigroup.[15] Let X the set of letters be called the *alphabet.* A *word* over the alphabet X is a finite sequence of elements of X: $(x_1, x_2, \ldots, x_n)$, $x_i \in X$. The empty word is denoted by λ. X^+ is the set of all non-empty finite words. X^+ is a semigroup under the operation of concatenation, it is called the *free semigroup.* $X^* = X^+ \cup \{\lambda\}$ is the *free monoid.*

A word $v \in X^*$ is a *factor* of a word $z \in X^*$ if there exist words $u, w \in X^*$ such that $z = uvw$. v is a *left factor* of z if there exists a word $w \in X^*$ such that $z = vw$. A word w is *primitive* if it is not a power of another word. For any nonempty word w, the smallest factor u such that $w = u^n$, $n \geq 1$ is the *primitive root* of w. We also use the notation $u = \sqrt{w}$.

2.2 Finite State Automata

By a finite state *automaton*, we mean a triple $\mathcal{A} = (A, X, \delta)$ where A is the (finite nonempty) *state set*, X is the *input alphabet* and $\delta : A \times X \to A$ is the *transition function.* We do not explicitly consider the output of the automaton as it can be recovered from the state and the input symbol. We tacitly use the state as the output.

We can naturally extend the transition function for words i.e. sequences of input symbols: for the empty word $\delta(a, \lambda) = a$, and for arbitrary words $u, v \in X^*$, $\delta(a, uv) = \delta(\delta(a, u), v)$. There is a natural equivalence relation, the *congruence induced by* $\mathcal{A}$ on words $u \equiv v$ if $\delta(a, u) = \delta(a, v)$ $\forall a \in A$, i.e. identifying words with

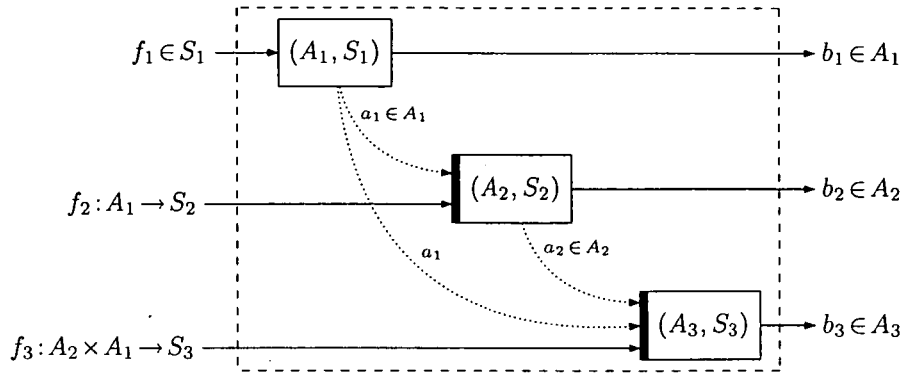


Figure 1: State transition in the wreath product $(A_3, S_3) \wr (A_2, S_2) \wr (A_1, S_1)$. The transformation (f_3, f_2, f_1) is applied to state (a_3, a_2, a_1) yielding $(b_3, b_2, b_1) = (a_3 \cdot f_3(a_2, a_1), a_2 \cdot f_2(a_1), a_1 \cdot f_1)$. The black bars denote the applications of functions f_2, f_3 according to hierarchical dependence. Note that the applications of these functions happen exactly at the same moment since their arguments are the previous states of other components, therefore there is no need to wait for the other components to calculate the new states. We use the state as the output of the automaton.

the same action on A. The *characteristic semigroup* $S(\mathcal{A})$, also called the *semigroup of the automaton*, is the set equivalence classes $X^+/\equiv$ of this congruence, with associative operation induced by concatenation. With the characteristic semigroup we can handle an automaton $\mathcal{A}$ as a transformation semigroup $(A, S(\mathcal{A}))$. Conversely if S is a semigroup then the corresponding automaton is $\mathcal{A}_S = (S^1, S)$, where the transition function is the right action of S on S^1.

An automaton $\mathcal{A}$ *emulates* another one $\mathcal{B}$ with states B if every computation which can be done in $\mathcal{B}$ can be done in $\mathcal{A}$ as well, i.e. $(B, S(\mathcal{B}))$ divides $(A, S(\mathcal{A}))$.

Using automata terminology constant mappings in transformation semigroups are often called *resets*. A *permutation-reset* automaton is an automaton such that each of its inputs acts either as a permutation or a constant map on states.

The *state transition graph* $D(\mathcal{A})$ of an automaton $\mathcal{A} = (A, X, \delta)$ is a digraph with A as the set of vertices and (a, x, b) is a labelled edge if $a \cdot x = b$, where $a, b \in A$, $x \in X$. It is a *loop-edge* if $a = b$. A *path* is a sequence of edges (a_i, x_i, b_i) $1 \leq i \leq n$ with $a_{i+1} = b_i$ for all $1 \leq i < n$, and the *label* of the path is $x_1 \ldots x_n$. A *loop* is a path with $b_n = a_1$.

2.3 Wreath Product Explained

Although the concept of the wreath product is not so complicated, it is not as easy to present the intuitive idea how the loop-free cascaded product works. After reading the formal definition a figure may shed light on how state transitions happen in the product (Fig. 1). It is also a great help first to consider a simpler product

with no dependence between the components.

Let $(A_n, S_n), \ldots, (A_1, S_1)$ be transformation semigroups called *components*. The indices $1, \ldots, n$ are called *coordinates*. The *direct product* $(A_n, S_n) \times \ldots \times (A_1, S_1)$ is the ts $(A_n \times \ldots \times A_1, S_n \times \ldots \times S_1)$ with the componentwise action

$$(a_n, \ldots, a_1) \cdot (s_n, \ldots, s_1) = (a_n \cdot s_n, \ldots, a_1 \cdot s_1).$$

Direct product is also called *parallel composition* as the components' state transitions do not depend on each other, and the order of the components does not really matter up to isomorphism.

Now we introduce an order-dependent connection between the components. Let $A = A_n \times \ldots \times A_1$ and $\mathcal{T}_A$ the full ts on A. Let S be the subsemigroup of $\mathcal{T}_A$ consisting of all transformations $s : A \to A$ satisfying the condition of *hierarchical dependence* of coordinates. Denote $p_k : A \to A_k$ the kth projection map, then for each $k = 1, \ldots, n$ there exists $f_k : A_{k-1} \times \cdots \times A_1 \to S_k$ such that

$$p_k\big((t_n, \ldots, t_{k+1}, t_k, \ldots, t_1) \cdot s\big) = t_k \cdot f_k(t_{k-1}, \ldots, t_1) = t'_k$$

$$\text{where } s \in S,\ \ t_k, t'_k \in A_k,\ \ k = 1, \ldots, n.$$

That is, the new kth coordinate t'_k resulting from the action of s depends only on the values of the old first k coordinates and on the transformation s. Moreover, it is given by acting with an element of S_k which depends only on s and $(t_{k-1}, \ldots, t_1)$. We can write this transformation as the ordered list of these functions: $s = (f_n, \ldots, f_1)$.

Then the transformation semigroup $(A, S) = (A_n, S_n) \wr \ldots \wr (A_1, S_1)$ is the *wreath product* of transformation semigroups $(A_n, S_n), \ldots, (A_1, S_1)$. Reading from left to right the last component is the top level of the hierarchy.

3 Holonomy Decomposition Theorem

The holonomy decomposition originates from Zeiger's method of proving the Krohn-Rhodes Theorem [16, 17, 7]. This algorithm work by the detailed study of how the semigroup S of an automaton (A, X, δ) acts on subsets of A. It looks for groups induced by S permuting some set of subsets of A. These groups are called the *holonomy groups*. These groups are the building blocks for the components of the decomposition. As we go deeper in the hierarchy of the cascade composition we have components that act on subsets with smaller cardinality.

The sketch of the algorithm to obtain a decomposition: First calculate the set of images of transformations in S. From now on, let $\mathcal{I}$ denote this set extended by A itself and its singletons. On $\mathcal{I}$ there is a preorder relation called *subduction* defined. A subset P is subduction related to a subset Q if P is contained in a resulting set of acting by some $s \in S$ on Q, i.e. $P \subseteq Q \cdot s$. The mutual relation of elements induces an associated equivalence relation $P \equiv Q \iff P \leq Q$ and $Q \leq P$. The set of equivalence classes are partially ordered by the subduction relation. The set of equivalence classes and their partial order are called the *subduction picture*. The

tiles B_P of a subset P ($P \in \mathcal{I}, |P| > 1$) are its proper subsets directly below it in the subduction preorder. The union of its tiles equals to P. The length of a longest strict path from a singleton to a subset P in the partial order of subduction equivalence classes defines the *height* of the subsets within the equivalence class of P. Equivalence classes with the same height are on the same hierarchical level. The sets of tiles for each element $Q \in \mathcal{I}$ form the *tiling picture*. The holonomy group H_Q of Q is the group (arising from elements of S^1) permuting the tile set B_Q of Q. The component $\overline{\mathcal{H}_i}$ of one hierarchical level i is the direct product of the holonomy groups belonging to the representative elements of equivalence classes with height i augmented with the constant mappings.

Theorem 1 (Holonomy Decomposition [6, 4]). *Let (A, S) be a finite transformation semigroup then (A, S) divides a wreath product of its holonomy permutation-reset transformation semigroups $(\mathcal{B}_1, \overline{\mathcal{H}_1}) \wr \cdots \wr (\mathcal{B}_h, \overline{\mathcal{H}_h})$.*

This strong formulation of part of the Krohn-Rhodes theorem is slightly different from the original since the components here are groups extended with constants and not simple groups and the divisors of the flip-flop. But these permutation-reset components can be easily decomposed into flip-flops and groups. Moreover the groups can be further decomposed into a series of simple groups using the Lagrange Coordinate Decomposition Theorem and Jordan-Hölder Theorem [8, 4]. Note that the top level of the hierarchy is the component with highest index, not 1.

4 Cycles in Automata

Definition 2. *A* graphical cycle *in an automaton (A, X, δ) is a cycle in its state transition digraph together with a word $w \in X^+$, i.e. a sequence of states $a_1, \ldots, a_n$ $n \geq 2$, where the states in the sequence are pairwise distinct except $a_1 = a_n$, and $w = x_1 \ldots x_{n-1}$, $x_i \in X$ such that $a_i \cdot x_i = a_{i+1}$ for all $1 \leq i \leq n-1$. The word $w = x_1 \ldots x_{n-1}$ is called the* label *of the cycle.*

Since $n \geq 2$ a loop edge is not a graphical cycle, and also, since $a_i \neq a_{i+1}$ within a graphical cycle, loop edges are not allowed.

Definition 3. *An* algebraic cycle *in an automaton $\mathcal{A} = (A, X, \delta)$ is a permutation group $(\{a_1, \ldots, a_n\}, \langle w \rangle)$ for which $a_i = a_j \Rightarrow i = j$, $n > 1$, and w is a word in X^+ such that $a_i \cdot w = a_{i+1}$ for all $1 \leq i < n$, and $a_n \cdot w = a_1$.*

The word w generates a cyclic group which acts faithfully on $\{a_1, \ldots, a_n\}$ by permutations. (Of course $\langle w \rangle$ might not act by permutations on A.) Obviously w^n is the identity element. Moreover, n being greater than 1 excludes trivial one-element groups. Note that loops are not generally algebraic cycles. The *generator* of the algebraic cycle is w, and its label is w^n.

5 Graphically Cycle-Free Automata

Definition 4. *An automaton is* graphically cycle-free *if it does not have any graphical cycle.*

The very simple structure of graphically cycle-free automata is reflected in their subduction pictures in the following way:

Lemma 5. *(A, S) is graphically cycle-free iff on every height level in each subduction relation equivalence class there is only one element.*

Proof: Let $P, Q \in \mathcal{I}$ and $P \equiv Q$ but $P \neq Q$. Since P, Q are finite $|P| = |Q|$. Clearly by finiteness there is at least one $x \in Q$ such that $x \notin P \cap Q$, otherwise P, Q would be the same. Due to the equivalence of P and Q we have $s, t \in S$ bijective mappings such that $P = Q \cdot s$ and $Q = P \cdot t$ and thus $(st)^n$ is the identity on Q for some $n > 0$, by the finiteness of P, Q. Since $x \cdot s = x' \neq x$ while $x \cdot (st)^n = x$, there must be a graphical cycle.

Conversely, a graphical cycle ensures the existence of an equivalence class with at least two elements at height zero. □

Another way to think about the proof of this lemma is to recognize that for the singleton subsets of the state set (at height zero) the equivalence classes are exactly the strongly connected components of the automaton's state transition graph.

This result can be exploited in the decomposition algorithm since if the equivalence classes are detected to all be singleton classes, then there is no need to look for holonomy groups at all and the holonomy identity-reset ts's can be built immediately.

6 Algebraically Cycle-Free Automata

It is a well-known result of algebraic automata theory that the star-free rational languages are recognized by exactly those automata whose characteristic monoid is aperiodic (having no nontrivial subgroup)[14]. It is also known that deciding aperiodicity for a finite automaton is PSPACE-complete[2]. We are interested in this problem for certain derived automata that arise naturally in the holonomy decomposition.

Intuitively one might expect that the state transition graph of an aperiodic automaton contains no cycles at all, but this is not true in general: there might be graphical cycles in it, while remaining aperiodic (see Fig 2). But with another type of cycles the notion of aperiodicity can be expressed.

Definition 6. *An automaton $\mathcal{A} = (A, X, \delta)$ is* algebraically cycle-free *if it does not have any algebraic cycle.*

The property of algebraic cycle-freeness is tied up with the primitivity of words, which act on some states as the identity.

Lemma 7. *An automaton $\mathcal{A} = (A, X, \delta)$ is algebraically cycle-free iff for all states $a \in A$ and for all words $w \in X^+$ such that $a \cdot w = a$, one of the following statements holds.*

1. *w is primitive.*
2. *w is not primitive but has primitive root $u \in X^+$, i.e. $w = u^n$, and $a \cdot u = a$.*

Proof: If w is primitive, then we are done. Otherwise $w = u^n$ where u is primitive. Let's suppose indirectly that $a \cdot u \neq a$. Let k be the least integer that $a \cdot u^k = a$ $(1 < k \leq n)$. Then $(\{a, a \cdot u, \ldots, a \cdot u^{k-1}\}, \langle u \rangle)$ is a cyclic permutation group (with at least two elements), therefore we have an algebraic cycle, contradicting our assumptions.

The converse is obvious due to the fact that a trivial permutation group does not constitute an algebraic cycle, and the conditions $1-2$ allow only trivial permutation groups. □

Remark 8. *Obviously Lemma 7 holds even if $a \cdot z \neq a$ for some left factor z of w.*

It is clear that in the absence of graphical cycles there cannot be any algebraic cycle. Thus,

Proposition 9. *If an automaton is graphically cycle-free then it is algebraically cycle-free.*

Now we show that aperiodic automata are exactly the algebraically (not the graphically) cycle-free ones.

Theorem 10. *The following are equivalent for an automaton $\mathcal{A} = (A, X, \delta)$ with corresponding transformation semigroup (A, S):*

1. *$\mathcal{A}$ is algebraically cycle-free.*
2. *S is aperiodic.*
3. *Holonomy groups are trivial for (A, S).*

Proof: (1) $\Rightarrow$ (2): Suppose S is not aperiodic, then we have a cyclic group $\langle v \rangle$ in S of order $n \geq 2$, where $v \in X^+$ is a word representing the generator. Thus v^n is the identity of the cyclic group, $v \equiv v^{n+1}$ and $v \not\equiv v^2$. Therefore $\exists a$ such that $a \cdot v \neq a \cdot v^2$ and $a \cdot v = a \cdot v^{n+1}$. Let $a' = a \cdot v$, thus $a' \cdot v^n = a'$ and since $\mathcal{A}$ is algebraically cycle-free we can apply Lemma 7: let $u = \sqrt{v^n} = \sqrt{v}$, then we have $a' \cdot u = a'$, $a' \cdot v = a'$ and finally $a \cdot v^2 = a \cdot v$, which is a contradiction.

(2) $\Rightarrow$ (1): For the converse we use again an indirect proof: Suppose there is an algebraic cycle, i.e. $(\{a_1, \ldots, a_n\}, \langle w \rangle)$ is a permutation group with $a_i \in A, w \in X^+$ and $n > 1$. Therefore $\mathbb{Z}_n$, the cyclic group with n elements, divides S. This cannot happen when S is aperiodic.

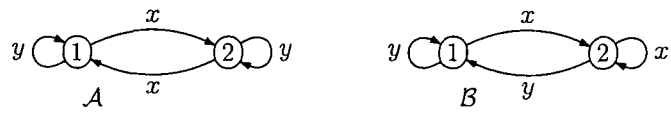


Figure 2: Automaton $\mathcal{A}$ has an algebraic cycle $(\{1,2\},\langle a\rangle)$. Automaton $\mathcal{B}$ has graphical cycles ab, ba, but they are labelled with primitive words.

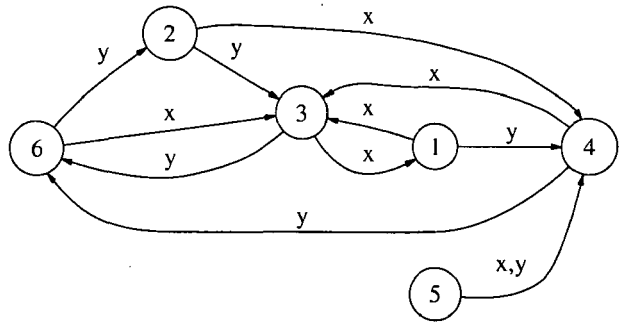


Figure 3: An automaton $\mathcal{A}$ with state set $A = \{1,2,3,4,5,6\}$ and alphabet $\{x,y\}$, where x and y are transformations with $x = (3\ 4\ 1\ 3\ 4\ 3)$, $y = (4\ 3\ 6\ 6\ 4\ 2)$.

(2) $\Leftrightarrow$ (3): The components of the holonomy decomposition are all divisors of the original semigroup, thus aperiodic semigroups have only trivial holonomy groups, and wreath products and divisors of aperiodic transformation semigroups are aperiodic. □

Corollary 11. *An automaton $\mathcal{A} = (A, X, \delta)$ is aperiodic if and only if*

$$\forall a \in A,\ w \in X^+,\ x \cdot w = a \Rightarrow a \cdot \sqrt{w} = a.$$

The distinction between algebraically cycle-free aperiodic and nonaperiodic automata is rather subtle. Two automata having the same state-transition graphs regarding their connectivity might belong to different classes depending on how the input symbols act on the state set (Fig. 2).

7 Non-Aperiodic Automata

A main concern of the holonomy decomposition is to find the nontrivial holonomy groups. Fortunately the tiling picture provides tools for locating the elements of $\mathcal{I}$ for which there exist nontrivial holonomy groups.

Lemma 12. *For an element Q of $\mathcal{I}$ in the tiling picture of (A,S) if there is a nontrivial holonomy group H_Q, then in its set of tiles B_Q there are at least two distinct tiles t_1, t_2 such that $t_1 \equiv t_2$.*

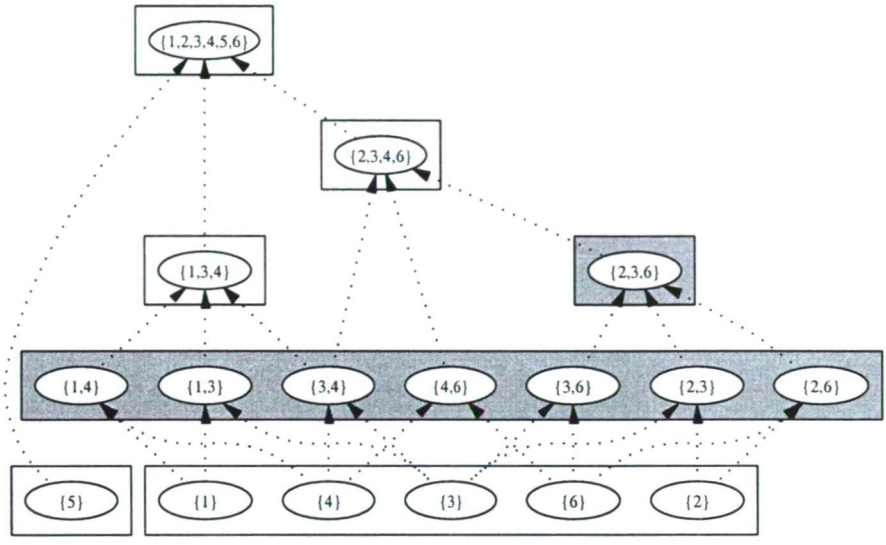


Figure 4: The tiling picture of automaton $\mathcal{A}$ in Fig. 3. The equivalence classes are denoted by boxes. Equivalence classes with elements having nontrivial holonomy groups are shaded. Dotted edges denote the 'tile of' relation.

Proof: H_Q being nontrivial means that there are some pair(s) of tiles for which there are transformations permuting them and thus they are mutually subduction related. □

The converse is not generally true as we can see in the example of an automaton (Fig 3) with tiling picture (Fig 4). For a trivial H_Q the set of tiles B_Q may contain distinct equivalent tiles, see Fig 5. In order to determine whether we have a nontrivial holonomy group for a $Q \in \mathcal{I}$ we define an extended automaton and examine its cycle structure. Denote the equivalence classes of subduction relation by E_1 to E_N.

Lemma 13. *If $P \in E_i$ and for some $s \in S$, $P \cdot s = Q$ such that $Q \notin E_i$ (leaving the equivalence class) then there is no transformation $t \in S$ such that $Q \cdot t \in E_i$ (no way back to the original equivalence class).*

Proof: Suppose there is such a t that $Q = P \cdot s$ and $P' = Q \cdot t$ with $P \equiv P'$. Due to the equivalence we have $P = P' \cdot s''$ for some $s'' \in S$, therefore $Q \cdot (ts'') = P' \cdot s'' = P$, thus $Q \equiv P$, which contradicts the original assumption that we leave the equivalence class of P. □

Let's define E_Q as the union of equivalence classes which contain at least one tile of $Q \in \mathcal{I}$. Formally: $E_Q = \bigcup_{E_i \cap B_Q \neq \emptyset} E_i$. Then the *tile automaton* of Q is

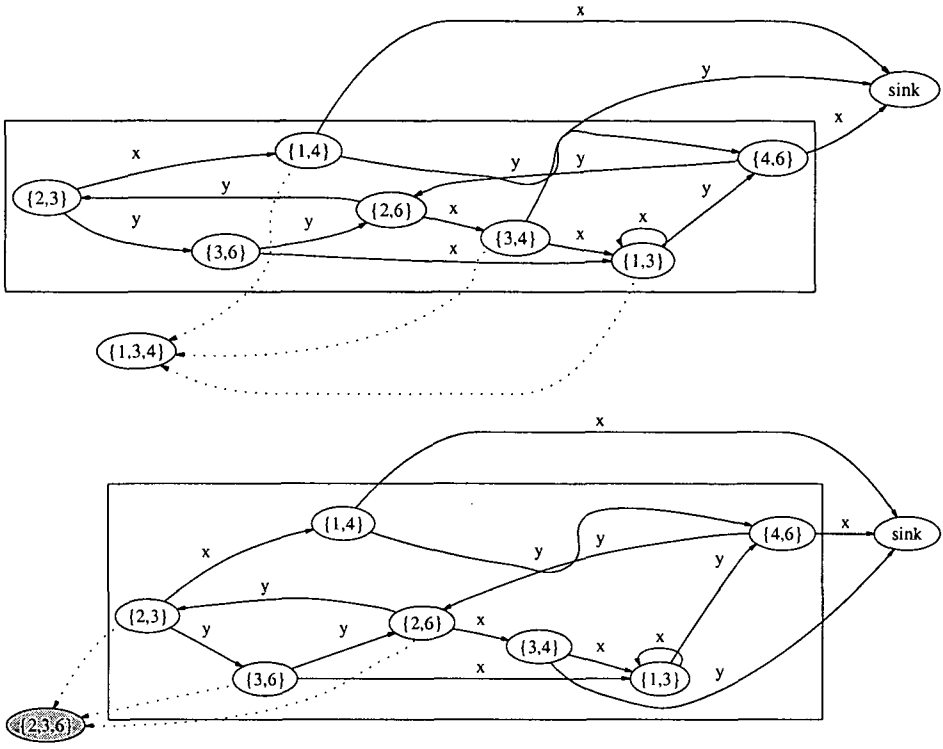


Figure 5: Two tile automata of automaton $\mathcal{A}$ in Fig. 3. $\mathcal{A}_{\{1,3,4\}}$ is trivial, while $\mathcal{A}_{\{2,3,6\}}$ is nontrivial with generator word y.

defined as $\mathcal{A}_Q = (E_Q \cup \{\varsigma\}, X, \gamma)$, where ς is a sink state, the input alphabet X is the same as the original automaton's, and γ is the natural extension of δ to act on subsets of A providing that if the image is not in some E_i then it is ς. This way ς represents going to another equivalence class not contained in E_Q, but according to Lemma 13 this can be represented as a sink since there is no way to come back.

The equivalence classes in E_Q form strongly connected components in $\mathcal{A}_Q$. When determining the nontriviality of H_Q we look for algebraic cycles in these components. We look not simply for independent algebraic cycles in each component as a word of a cycle might not permute the tile elements in another component, but for parallel algebraic cycles. This way we can recast the characterization of a holonomy group element in terms of algebraic cycles. More formally:

Proposition 14. *H_Q is nontrivial iff there exists a word $w \in A^+$ and B_Q can be partitioned into $\{T_1, \ldots, T_k\}$ subsets such that either*

1. T_i *consists of exactly one tile and* $T_i \cdot w = T_i$, *or*
2. $T_i \cdot \langle w \rangle \subseteq B_Q \cap E_j$ *for some* $1 \leq j \leq N$, *and* $(T_i \cdot \langle w \rangle, \langle w \rangle)$ *is an algebraic cycle in* $\mathcal{A}_Q$

holds for all T_i, $1 \leq i \leq k$, *and* (2) *must hold for at least one* T_i.

In short the proposition characterizes when the transformation induced by w nontrivially permutes B_Q. This transformation is clearly a nontrivial holonomy group element. From Lemma 13, $T_i \cdot w^n \in (B_Q \cap E_j)$ follows for any $n \geq 0$. Therefore the algebraic cycles contained in B_Q generated by w are all disjoint. If all intersections $(B_Q \cap E_j)$ are singletons, or none of them contains an algebraic cycle then H_Q is trivial. This fact can be exploited in efficient decomposition algorithms of the holonomy decomposition by excluding cases where the construction of the holonomy group should not be attempted.

8 Conclusions

Using an implementation of the holonomy decomposition we could get new insights about its working mechanism and found a relation between the cycle structure of an automaton and its holonomy components. We also showed that detecting cycles with primitive words helps in excluding elements of $\mathcal{I}$ when searching for holonomy groups. Currently we are investigating the possibility of efficient construction of holonomy groups by using the extended tile automata replacing the current algorithm which is based on a breadth-first search in the space of semigroup elements. These results will eventually lead to improvements of the decomposition algorithms providing efficient and scalable tools for attacking real-world problems such as analyzing metabolic networks [13], understanding biological complexity [12], AI applications [11] and so on.

References

[1] Michael A. Arbib, editor. *Algebraic Theory of Machines, Languages, and Semigroups.* Academic Press, 1968.

[2] Sang Cho and Dung T. Huynh. Finite-automaton aperiodicity is pscpace-complete. *Theoretical Computer Science*, 88:99–116, 1991.

[3] A.H. Clifford and G.B. Preston. *The Algebraic Theory of Semigroups (Mathematical Survey, No 7)*, volume 1 of *Mathematical Survey.* American Mathematical Society, 2nd edition, 1967.

[4] Pál Dömösi and Chrystopher L. Nehaniv. *Algebraic Theory of Finite Automata Networks: An Introduction*, chapter 3, The Krohn-Rhodes and Holonomy Decomposition Theorems. SIAM Series on Discrete Mathematics and Applications, 2005.

[5] Attila Egri-Nagy and Chrystopher L. Nehaniv. GrasperMachine, Computational Semigroup Theory for Formal Models of Understanding. (`http://graspermachine.sf.net`)., 2003.

[6] Samuel Eilenberg. *Automata, Languages and Machines*, volume B. Academic Press, 1976.

[7] Abraham Ginzburg. *Algebraic Theory of Automata.* Academic Press, New York, 1968.

[8] Marshall Hall. *The Theory of Groups.* The Macmillan Company, New York, 1959.

[9] Kenneth Krohn and John Rhodes. Algebraic theory of machines. I. Prime decomposition theorem for finite semigroups and machines. *Transactions of the American Mathematical Society*, 116:450–464, April 1965.

[10] Chrystopher L. Nehaniv. Algebraic engineering of understanding: Global hierarchical coordinates on computation for the manipulation of data, knowledge, and process. In *Proc. 18th Annual International Computer Software and Applications Conference (COMPSAC 94)*, pages 418–425. IEEE Computer Society Press, 1994.

[11] Chrystopher L. Nehaniv. Algebra and formal models of understanding. In Masami Ito, editor, *Semigroups, Formal Languages and Computer Systems*, volume 960, pages 145–154. Kyoto Research Institute for Mathematics Sciences, RIMS Kokyuroku, August 1996.

[12] Chrystopher L. Nehaniv and John L. Rhodes. The evolution and understanding of hierarchical complexity in biology from an algebraic perspective. *Artificial Life*, 6:45–67, 2000.

[13] John L. Rhodes. *Applications of Automata Theory and Algebra with the Mathematical Theory of Complexity to Finite-State Physics, Biology, Philosophy, Games, and Codes.* book submitted for publication.

[14] M. P. Schützenberger. On finite monoids having only trivial subgroups. *Information and Control*, 8:190–194, 1965.

[15] H. J. Shyr. *Free monoids and languages.* Hon Min Book Company, Taichung, Taiwan, 2001.

[16] H. Paul Zeiger. Cascade synthesis of finite state machines. *Information and Control*, 10:419–433, 1967. plus erratum.

[17] H. Paul Zeiger. Yet another proof of the cascade decomposition theorem for finite automata. *Math. Systems Theory*, 1:225–228, 1967. plus erratum.

Acta Cybernetica **17** (2005) 213–224.

A Hierarchical Evaluation Methodology in Speech Recognition

Gábor Gosztolya* and András Kocsor*

Abstract

In speech recognition vast hypothesis spaces are generated, so the search methods used and their speedup techniques are both of great importance. One way of getting a speedup gain is to search in multiple steps. In this multi-pass search technique the first steps use only a rough estimate, while the latter steps apply the results of the previous ones. To construct these raw tests we use simplified phoneme groups which are based on some distance function defined over phonemes. The tests we performed show that this technique could significantly speed up the recognition process.

Keywords: speech recognition, search methods, multi-stack decoding, multi-pass search, phoneme grouping.

1 Introduction

Automatic speech recognition (ASR) is a pattern classification problem [1] in which a continuously varying signal has to be mapped to a string of symbols (the phonetic transcription). Besides the identification of speech segments with grammatical phonemes [2], efficient searching in the induced hypothesis space [3] is of great importance as well. This work is related to both areas: first we give a hierarchical scheme of the Hungarian phonemes, then we try to exploit this structure in the search process.

In this paper we want to construct a multi-pass search method where the different steps are based on the selection of the different phoneme groups used. However this construction of the phoneme groups is not trivial, so the choice of the algorithm we use heavily affects the speed and recognition accuracy of the speech recognition system.

The structure of this paper is as follows. First we define the speech recognition problem and the search task. Then we construct a phoneme grouping method based on a distance function between phonemes. Lastly, after presenting and analyzing the test results, we mention some suggestions for future study.

*Research Group on Artificial Intelligence of the Hungarian Academy of Sciences and University of Szeged, H-6720 Szeged, Aradi vértanúk tere 1., Hungary, e-mail: {ggabor,kocsor}@inf.u-szeged.hu

2 Search Spaces in Speech Recognition

In speech recognition problems we have a speech signal represented by a series of observations $A = a_1 a_2 \dots a_t$, and a set of possible phoneme sequences (words or word sequences) which will be denoted by W. Our task is to find the word $\hat{w} \in W$ defined by

$$\hat{w} = arg \max_{w \in W} P(w|A), \tag{1}$$

which, using Bayes' theorem, is equivalent to the following maximization problem:

$$\hat{w} = arg \max_{w \in W} \frac{P(A|w) \cdot P(w)}{P(A)}. \tag{2}$$

Further, noting the fact that $P(A)$ is the same for all $w \in W$, we have that

$$\hat{w} = arg \max_{w \in W} P(A|w)P(w). \tag{3}$$

Speech recognition models can be divided into two types (the discriminative and generative ones), depending on whether they use Eq. (1) or Eq. (3). Throughout this paper we will apply the customary, generative approach [4].

Unified view

Both the generative and discriminative models exploit *frame-based* and/or *segment-based* [5] features, and this fact allows us to have a unified framework of the frame- and segment-based recognition techniques. To make this clearer, we will provide a brief outline of this framework along with the hypothesis structure that will be generated.

Now let us commence with some definitions. Let us define w as $o_1 o_2 \dots o_n$, where o_j is the jth phoneme of word w. Furthermore, let $A_1, A_2, \dots, A_n$ be non-overlapping segments of the observation series $A = a_1 a_2 \dots a_t$, where $A_j = a_{t_{j-1}} \dots a_{t_j}$, $j \in \{1, \dots, n\}$. An A_j segment is defined by its start and end times and will be denoted by $[t_{j-1}, t_j]$. For a segmentation $A = A_1, A_2, \dots, A_n$ we put the values of the time indices corresponding to each segment into a vector $T_n = [t_0, t_1, \dots, t_n]$ $(1 = t_0 < t_1 < \dots < t_n = t)$. We make the conventional assumption that the phonemes in a word are independent so that $P(A|w)$ can be obtained from $P(A_1|o_1), P(A_2|o_2), \dots, P(A_n|o_n)$ in some way. To calculate $P(A|w)$, various aggregation operators can be used at two distinct levels. In the first one the $P(A_j|o_j)$ probability values are supplied by a g_1 operator, i.e.

$$P(A_j|o_j) = g_1([t_{j-1}, t_j], o_j),$$

which provides an overall value for measuring how well the A_j segment represents the o_j phoneme based on local information sources. In the second one, another operator (g_2) is used to construct $P(A|w)$ using the probability values $P(A_1|o_1), \dots, P(A_n|o_n)$.

Frame-based approach

The well-known *Hidden Markov Model (HMM)* [6] is basically a frame-based approach, i.e. it handles a speech signal frame by frame. Usually a *Gaussian Mixture Model (GMM)* is applied to compute the $P(a_l|o_j)$ values (for delta and delta-delta features neighboring observations are also required) and for the A_j segment the $g_1([t_{j-1}, t_j], o_j)$ value is defined by

$$\prod_{l=t_{j-1}}^{t_j} c_{o_j} \cdot P(a_{l-k} \dots a_{l+k}|o_j), \tag{4}$$

where $0 \leq c_{o_j} \leq 1$. Practically speaking, g_1 includes all the information we have when we are in a particular state of a HMM model. We note here that, instead of GMM, Artificial Neural Networks (*ANNs*) and other machine learning algorithms that can be used for density estimation are also viable. This alternative provides a way for creating model hybrids. As for the $P(A|w)$ value, the g_2 operator is defined by

$$P(A_n|o_n) \prod_{j=1}^{n-1} (1 - c_{o_j}) P(A_j|o_j). \tag{5}$$

Segment-based approach

In the segment-based speech recognition approach – like the SUMMIT system of MIT [7] or our OASIS [8] – g_1 will usually be the direct output of some machine learning algorithm using features that describe the whole $[t_{j-1}, t_j]$ segment. Among the many possibilities the most conventional choice of g_2 is simply to multiply the probabilities, but in earlier works we showed that using other operators can be beneficial for both the speed and performance [9]. In the following we will stick to multiplication, but the improvements discussed here could also be implemented using other aggregation operators.

The hypothesis space

The task of speech recognition is essentially a selection problem over a Cartesian product space where the first dimension is a set of word hypotheses, while the second is a set of segmentations. Given a set of words W, we use $Pref_k(W)$ to denote the k-long prefixes of all the words in W having at least k phonemes. Let

$$T^k = \{[t_0, t_1, \dots, t_k] : 1 = t_0 < t_1 < \dots < t_k \leq t\} \tag{6}$$

be the set of sub-segmentations made of k segments over the observation series $a_1 a_2 \dots a_t$. The hypotheses will be object pairs, i.e. they are elements of

$$H = \bigcup_{k=0}^{\infty} (Pref_k(W) \times T^k). \tag{7}$$

We will denote the root of the tree – the initial hypothesis – by $h_0 = (\emptyset, [t_0])$ $(h_0 \in H)$. Here $Pref_1(W) \times T^1$ will contain the first-level nodes. For a $(o_1 o_2 \ldots o_j, [t_0, \ldots, t_j])$ leaf we link all $(o_1 o_2 \ldots o_j o_{j+1}, [t_0, \ldots, t_j, t_{j+1}]) \in Pref_{j+1}(W) \times T^{j+1}$ nodes.

Now we need to evaluate the nodes of the search tree. To this end let the g_1 and g_2 functions be defined by some aggregation operators. Then, for a node $(o_1 o_2 \ldots o_j, [t_0, \ldots, t_j])$, the value is defined by

$$g_2(g_1([t_0, t_1], o_1), \ldots, g_1([t_{j-1}, t_j], o_j)). \tag{8}$$

Note that, in practice, it is worth calculating this expression recursively. After defining the evaluation methodology we will look for a leaf with the highest probability.

This definition in typical circumstances leads to a huge hypothesis space, where a full search will be impractical because of the big run time and memory requirements. This leads us to employ heuristics like the well-known Viterbi beam search [10] or our choice, the multi-stack decoding algorithm [11].

3 Clustering the Phoneme Set

In this section we discuss the technique we used to create smaller, more compact phoneme groups. First we define two novel, similar functions between phonemes, prove that they have the right sort of properties to be distance functions, then utilize them in the phoneme-clustering problem.

There is no simple answer to the problem of how we should construct the phoneme groups mentioned above. We might base it on previous grammatical knowledge or use the confusion matrix of the phoneme classifier. The justification for the latter option is that the recognition process is already heavily based on the phoneme classifier.

A classifier gets some set of observations, and its task is to classify this set into one of the $\Omega = \{\omega_1, \omega_2, \ldots, \omega_K\}$ classes. A *confusion matrix* A is constructed in such a way that $a_{i,j}$ is the number of phonemes belonging to ω_j from a selected test set which we classified as ω_is by the classifier [12]. In our case the classifier is used to categorize the parts of speech into one of the phoneme classes. The confusion matrix of a good classifier is close to a diagonal matrix, which is why we will concentrate on the number of misclassified items (i.e. the number of examples that were incorrectly classified).

Grouping phonemes is a standard *clustering problem* [13]: some points (here, the phoneme classes) are to be assigned to a certain number of *clusters* (in our case, phoneme groups). There are some quite general algorithms for this task. The one we are going to use needs a distance function for two clusters, which will be defined below, but first we will explain how this algorithm works.

At the start each phoneme will be considered as different clusters. Then, in each step, we find those C_i and C_j clusters where $\mathcal{D}(C_i, C_j)$ is minimal, and combine

Table 1: An example of a confusion matrix

	1	2	3	4	5	6	7	8	9	10	11
1	2502	3	96	35	4	0	0	4	8	0	12
2	18	965	3	24	3	0	0	0	5	1	49
3	87	8	875	19	2	0	0	5	11	0	18
4	43	11	16	271	1	1	0	1	2	0	12
5	12	2	3	2	2250	257	80	101	53	5	48
6	0	1	0	0	51	299	17	22	8	24	17
7	1	0	0	0	46	31	208	6	1	15	5
8	3	4	3	1	70	39	8	5235	111	19	116
9	7	1	6	3	19	10	2	97	461	2	77
10	1	0	0	0	12	88	25	62	11	830	8
11	39	71	21	31	38	23	19	102	367	18	2316

them. We repeat this until $\mathcal{D}(C_i, C_j) \geq L$, where L is a parameter. (See Appendix A for the pseudocode of this algorithm.)

To define our novel distance functions first let A' be a normalized matrix for the confusion matrix A of the applied phoneme classifier. It takes the form

$$a'_{i,j} = a_{i,j} / \sum_k a_{k,j} \quad i, j \in \{1, \ldots, K\}.$$

We can assume that $\sum_k a_{k,j} \neq 0$, otherwise it would mean that the jth phoneme has no examples in the test database. Next we define a distance function based on this A' matrix. Let

$$d^1_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } a'_{i,j} = a'_{j,i} = 0 \text{ and } i \neq j \\ -log(a'_{i,j}) & \text{if } a'_{j,i} = 0 \text{ and } a'_{i,j} \neq 0 \\ -log(a'_{j,i}) & \text{if } a'_{i,j} = 0 \text{ and } a'_{j,i} \neq 0 \\ \min(-log(a'_{i,j}), -log(a'_{j,i})) & \text{otherwise,} \end{cases} \tag{9}$$

and let

$$d^2_{i,j} = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } a'_{i,j} = a'_{j,i} = 0 \text{ and } i \neq j \\ -log((a'_{i,j} + a'_{j,i})/2) & \text{otherwise.} \end{cases} \tag{10}$$

Now let D' be the output of some shortest path-finding algorithm with the input of the D^1 or D^2 matrix. (We can choose either of them, but of course if we use both, this choice leads to twice as many test cases. The figures we obtained can be seen in the results section.) D' is a distance function, moreover it satisfies the criteria of being a metric because

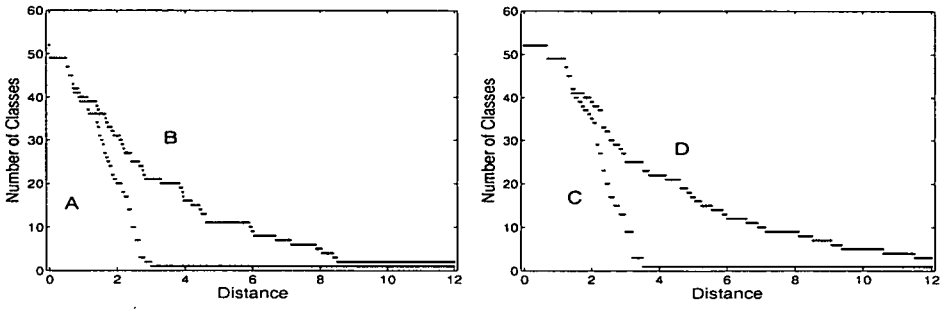


Figure 1: Number of phoneme groups (classes) – L limit diagram for the four distance-variations; d^1 and d^2, respectively. The A and C curves belong to $\mathcal{D}_{min}$, while the B and D curves belong to the $\mathcal{D}_{max}$ group distance function.

- $d'_{i,i} = 0$
- $d'_{i,j} = d'_{j,i}$
- $d'_{i,j} \leq d'_{i,k} + d'_{k,j}$

Now we have to define the distance $\mathcal{D}(C_i, C_j)$ of the clusters C_i and C_j, when we have only the $d'(x_i, y_i)$ values (the distance between different phonemes). To do this we have two straightforward options [13]:

$$\mathcal{D}_{min}(C_i, C_j) = min_{x,y}\{d'(x, y)|x \in C_i, y \in C_j\}, \tag{11}$$

and

$$\mathcal{D}_{max}(C_i, C_j) = max_{x,y}\{d'(x, y)|x \in C_i, y \in C_j\}. \tag{12}$$

The former tends to create longer, larger clusters, while the latter usually creates more compact ones. In our experiments we tested both versions.

We should mention here that the use of $\mathcal{D}_{max}$ in this algorithm could lead to a nondeterministic case if, at any given point, there exist some clusters C_i, C_j and C_k such that $\mathcal{D}_{max}(C_i, C_j) = \mathcal{D}_{max}(C_i, C_k)$. Note here that $\mathcal{D}_{min}$ is not a metric because in some cases the triangle inequality does not hold: there exist C_i, C_j and C_k clusters such that $\mathcal{D}_{min}(C_i, C_j) \nleq \mathcal{D}_{min}(C_i, C_k) + \mathcal{D}_{min}(C_k, C_j)$.

3.1 Tests

Applying the clustering algorithm (using one of the above $\mathcal{D}$ functions) will lead to a series of unions and a series of distance values. Based on them we can choose the possible values of the limit L, which will result in phoneme groups that will be used in the recognition process. Obviously, good L values are those where there is a nice gap between successive distance values in the output.

After examining Figure 1 we identified those bigger flat regions in each curve. For each of them we selected three Ls, resulting in the same number of phoneme

groups, which were later used in the multi-pass recognition method. The corresponding recognition steps were called Pass 1 (p_1), Pass 2 (p_2) and Pass 3 (p_3), with the number of phoneme groups varying from 27 to 34, from 17 to 21 and from 10 to 13, respectively. The default phoneme set was labelled p_0 and had 52 phonemes.

4 The Search Process

Given the phoneme groups – and hence the hypothesis space – we still have to search for the best hypothesis. There are standard search heuristics for this task, from which we chose the multi-stack decoding algorithm. Moreover, there is the possibility of constructing multi-pass methods where there are multiple steps in the search process. Here we decided to apply this idea using the already constructed phoneme groups.

Multi-pass Search Strategies

In general, **multi-pass methods** work in two or more steps: in the first pass the less likely hypotheses are discarded because of some condition requiring low computational time. Then, in the later passes, only the remaining hypotheses are examined by more complex, reliable evaluations, which will approximate the probabilities of the hypotheses more closely. (In the common search methods only the last pass remains, so more hypotheses are scanned there, making the process more time-consuming.)

To speed up the earlier steps, we need to construct faster phoneme classifiers, and the usual way of doing this is to reduce the number of features. (In our system, where ANNs are used, it also leads to a lower number of hidden neurons.) Here the number of phoneme groups was decreased. In the first pass a search with a very restricted phoneme set was performed. Then, in the later passes, more and more detailed phoneme groupings were used, where the dictionary consisted of the 'winning' words of the previous level. Obviously, during the last pass we had to use the original phoneme set to get only one word as a result, not a set of words. At each level we employed the multi-stack decoding algorithm in the search process.

The Multi-stack Search Method

The multi-stack decoding algorithm [11] is one of the heuristic search methods we mentioned earlier, and we chose this one as our basic search technique. To discuss the method first we have to give a definition. A *stack* is a structure for keeping hypotheses in. Moreover, we use limited-sized stacks: if there are too many hypotheses in a stack, we prune the ones with the highest cost.

In this algorithm we assign a separate stack to each time instance t_i and store the hypotheses in the stack according to their end times. In the first step we place h_0 (the initial hypothesis) into the stack associated with the first time instance, then, advancing in time, we pop each hypothesis in turn from the given stack, extend

them in every possible way, and put the new hypotheses into the stack associated with their new end times. Algorithm 2 in Appendix B shows the pseudocode for multi-stack decoding.

The multi-stack decoding algorithm has one parameter, the *stack size*. Decreasing it usually decreases the accuracy of the method (i.e. the recognition performance), while a greater stack size leads to increased run times and therefore a slower speech recognition system. Thus it is very important to find the best parameter value, which might mean a trade-off between accuracy and speed. Above a certain parameter setting there is no change in accuracy and this is often what we call the optimal value.

5 Experiments and Results

For testing purposes we used a corpus of 500 children uttering 60 words each, making a total of 30,000 utterances of 2000 different Hungarian words with a variance related to everyday-use occurrence. 24,000 utterances were used for training, while 6,000 words remained for testing purposes – including being the basis for phoneme-group generation. Many of the young speakers had just learned to read and some of them had difficulties with pronunciation, which led to a diverse database. Moreover, many of the words in the database (and thus, in the dictionary) were similar to each other with a phoneme difference of just one or two, which made the recognition quite difficult.

The phoneme recognition rate was 79.47% on the original phoneme set, and it remained around 80% when we applied the restricted phoneme sets. The diversity of the database led to a basic word recognition percentage of 84.14% with the OASIS system, whereas the *HTK* system we used for reference achieved a score of 84.34%. In our tests we expected a word accuracy of at least 80% for a multi-pass configuration.

5.1 Results

The speed of a multi-pass configuration was measured in the number of phoneme classifications averaged for a word. (We found that this was analogous to the actual running speed.) As the ANNs used were of unequal size at each step, the results were normalized to the speed of the phoneme classifier on the last pass. Moreover, for a multi-pass configuration, because at each level there is a multi-stack decoding algorithm used with a different parameter (stack size), there is room for adjusting both the speed and performance. In this case all levels were tested with different parameters, and the best configuration was the one which satisfied the accuracy criterion, and proved to be the fastest among these.

After considering the phoneme-clustering methods, we found we had four possibilities for selecting the phoneme groups and hence selecting the first passes of the multi-pass search method employed. In the table we show all four, providing a way of comparing them exhaustively. "•" means that we applied the given recognition

Table 2: Recognition results

Used passes				d^1 (minimum)		d^2 (average)	
p_0	p_1	p_2	p_3	$\mathcal{D}_{min}$	$\mathcal{D}_{max}$	$\mathcal{D}_{min}$	$\mathcal{D}_{max}$
•	o	o	o	12,578.01	12,578.01	12,578.01	12.578.01
•	•	o	o	6,697.17	6,656.59	4,641.99	5,390.84
•	o	•	o	5,784.14	6,682.20	–	7,831.17
•	o	o	•	–	–	–	**3,713.60**
•	•	•	o	7,727.77	5,477.20	–	7,286.60
•	•	o	•	–	–	–	6,647.32
•	o	•	•	–	–	–	5,062.17
•	•	•	•	–	–	–	6,463.53

pass in the configuration, while "o" denotes that this pass was omitted; a value "–" means that the given configuration could not attain the required recognition accuracy; d^1 and d^2 denote the chosen distance function between phonemes, while $\mathcal{D}_{min}$ and $\mathcal{D}_{max}$ denote the chosen distance function between phoneme groups, respectively.

Examining the results led us to the following observations. First, it is clear that it is possible to speed up a speech recognition system with a multi-pass search method by creating phoneme groups. Because the last pass is always executed using the original phoneme set and thus on the original phoneme classifier, a faster multi-pass search algorithm means that in the earlier passes the list of possible words was drastically reduced; thus the last pass was able to attain a good recognition accuracy even with small-sized stacks. Second, it seems that using $\mathcal{D}_{min}$ in the clustering algorithm leads to a worse result than $\mathcal{D}_{max}$. Third, we noticed that two-pass searches performed better than three- or four-pass configurations. The one-pass configuration was the slowest of the ones we tested when we wanted to achieve an accuracy of at least 80%.

6 Conclusion

In this paper we defined a novel speech recognition method which applied a hierarchical scheme of phoneme-group clusterings. We based this clustering on novel distance functions between phonemes. These functions, which characterized the phoneme set, employed the confusion matrix of the phoneme-classifying neural networks. Then the application of a well-known shortest path-finding algorithm supplied the final distance values, which formed the input for a general clustering algorithm. With this approach using increasingly detailed phoneme structures we were able to create a hierarchical speech recognition method. According to the test results the proposed hierarchical recognition method was able to significantly speed up the speech recognition process by a factor of 3 or 4. Also, our method is insensitive to the type of the phoneme-classifier, so various techniques can be used like C4.5 and GMM, which will be the subject of future work.

References

[1] C.M. Bishop, *Neural Networks for Pattern Recognition,* Clarendon Press, Oxford, 1995.

[2] A. Kocsor, L. Tóth, A. Kuba Jr., K. Kovács, M. Jelasity, T. Gyimóthy and J. Csirik, *A Comparative Study of Several Feature Space Transformation and Learning Methods for Phoneme Classification,* International Journal of Speech Technology, Vol. 3, Number 3/4, pp. 263-276, 2000.

[3] G. Gosztolya and A. Kocsor, *Improving the Multi-Stack Decoding Algorithm in a Segment-based Speech Recognizer,* Proceedings of the 16th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems, IEA/AIE 2003, LNAI 2718, pp. 744-749, Springer Verlag, 2003.

[4] F. Jelinek, *Statistical Methods for Speech Recognition,* The MIT Press, 1997.

[5] M. Ostendorf, V. Digalakis and O. A. Kimball, *From HMMs to Segment Models: A Unified View of Stochastic Modeling for Speech Recognition,* IEEE Transactions on Acoustics, Speech and Signal Processing, volume 4, pp. 360–378, 1996.

[6] L.Rabiner and B.-H. Juang, *Fundamentals of Speech Recognition* Prentice Hall, 1993.

[7] J. Glass, J. Chang, M. McCandless, *A Probabilistic Framework for Features-Based Speech Recognition,* Proceedings of International Conference on Spoken Language Processing, Philadelphia, PA, pp. 2277-2280, 1996.

[8] A. Kocsor, L. Tóth and A. Kuba Jr., *An Overview of the Oasis Speech Recognition Project,* Proceedings of ICAI '99, Eger-Noszvaj, Hungary, 1999.

[9] G. Gosztolya and A. Kocsor, *Aggregation Operators and Hypothesis Space Reductions in Speech Recognition,* Text, Speech and Dialogue '04, Brno, Czech Republic, 2004.

[10] P.E. Hart, N.J. Nilsson and B. Raphael, *Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths",* SIGART Newsletter, No. 37, pp. 28-29, 1972.

[11] L.R. Bahl, P.S. Gopalakrishnan, R.L. Mercer, *Search issues in large vocabulary speech recognition,* Proceedings of the 1993 IEEE Workshop on Automatic Speech Recognition, Snowbird, UT, 1993.

[12] R. O. Duda, P. E. Hart And D. G. Stork, *Pattern Classification,* John Wiley & Sons Inc., 2001.

[13] D. Hand, H. Mannila and P. Smyth, *Principles of Data Mining,* MIT Press, 2001.

Appendix A

The pseudocode of a general clustering algorithm. "$\leftarrow$" means that a variable is assigned a value. The parameters are $x_1, x_2, \ldots, x_n$, the initial points to be clustered (grouped), a $\mathcal{D}(C_i, C_j)$ distance function, and an L value for the stopping criterion.

Algorithm 1 General clustering algorithm

```
for i = 1, ..., n do
  C_i ← {x_i}
end for
while there is more than one cluster left do
  (i, j) ← arg max D(C_i, C_j) is minimal
  if D(C_i, C_j) > L then
    break
  end if
  C_i ← C_i ∪ C_j
  Remove C_j
end while
```

Appendix B

The multi-stack decoding pseudocode described by Algorithm 2. "$\Leftarrow$" means pushing a hypothesis into a stack. $Stack[t_i]$ means a stack belonging to the t_i time instance. A $H(w, T)$ hypothesis denotes a phoneme sequence and time-instance se-

Algorithm 2 Multi-stack decoding algorithm

```
Stack[t_0] ⇐ h_0(∅, [t_0])
for i = 0 ... n do
  while not empty(Stack[t_i]) do
    H(w, T) ← top(Stack[t_i])
    if t_i = t_max then
      return H
    end if
    for t_l = t_{i+1} ··· t_{i+maxlength} do
      for all {v | wv ∈ Pref_{1+length of w}} do
        H'(w', T ∪ t_l) ← extend H with v
        Stack[t_l] ⇐ H'
      end for
    end for
  end while
end for
```

quence pair. *Extending* a hypothesis $H(w,T) = H(w,[t_0,\ldots,t_k])$ with a phoneme v and a time t_i results in a hypothesis $H'(wv,T \cup t_i) = H'(wv,[t_0,\ldots,t_k,t_i])$, where the cost of the new hypothesis is calculated via the g_2 operator, after applying the g_1 function. Here we denote the maximal length of a phoneme by *maxlength*.

Acta Cybernetica 17 (2005) 225–245.

Optimal Trajectory Generation for Petri nets*

Szilvia Gyapay† and András Pataricza†

Abstract

Recently, the increasing complexity of IT systems requires the early verification and validation of the system design in order to avoid the costly redesign. Furthermore, the efficiency of system operation can be improved by solving system optimization problems (like resource allocation and scheduling problems). Such combined optimization and validation, verification problems can be typically expressed as reachability problems with quantitative or qualitative measurements. The current paper proposes a *solution to compute the optimal trajectories for Petri net-based reachability problems with cost parameters*. This is an improved variant of the basic integrated verification and optimization method introduced in [11] combining the efficiency of *Process Network Synthesis* optimization algorithms with the modeling power of *Petri nets*.

1 Introduction

As the quality of service delivered by IT systems becomes more and more crucial to production and the life of the society, their correct and efficient operation has to be proved already during the design phase. Validation and verification methods are known to assure the correctness of the services, while optimization may serve to calculate the quantitative performance characteristics of a system by estimating its quantitative boundaries and to minimize operation costs.

The fulfillment of the requirements have to be addressed already during the early design phases in order to avoid costly redesigns. Recently, system designers use the *Unified Modeling Language (UML)* that became the standard object-oriented modeling language since it provides a semi-formal, concise description of complex systems including the means to model both its static and dynamic behavior. UML can be extended to incorporate quantitative measures, requirements, and constraints.

While UML is a proper means for system and requirements modeling the analysis itself has to be carried with the help of some mathematical model analysis

*This work was supported by project OTKA T038027.

†Budapest University of Technology and Economics, Department of Measurement and Information Systems, Magyar tudósok körútja 2., H-1117, Budapest, Hungary. Phone: +36-1-463-3579, +36-1-463-3595 Fax: +36 1 463-2667 E-mail:{gyapay,pataric}@mit.bme.hu

tool. Recent research efforts aim at an automated transformation from UML models to mathematical analysis tools [27]. One of the most challenging problems in mathematical analysis of IT systems is the simultaneous analysis of the dynamical behavior of the system and its impact to the quantitative measures as it meets a combination of a mathematical paradigm describing the control logic of the application and the quantitative impact of it.

Petri nets are used in the design of IT systems either as a direct modeling paradigm or derived from engineering models. They describe the system logic, i.e. they define the potential behavior of the system by identifying the trajectories in the system state space in a compact form. However, the requirement is frequently not only to provide a correct behavior of the system, but to perform some functionality in an optimal way. Therefore, Petri nets first need to be extended with cost or time parameters. Then the reachability problem can be formulated as an *optimal trajectory problem*: to derive the optimal trajectory from the initial state to a given state. Unfortunately, the optimal trajectory problem has been solved only for rather restricted classes of Petri nets without practical relevance.

Our overall objective is to derive from the UML specification of the application and to estimate the worst case characteristics of a system upon *temporal constraints* on its operation sequence. (i) At first, an automated transformation on attributed graph grammars maps the UML models into Petri nets (discussed in [10]). (ii) Secondly, a basis of all firing sequences representing possible operations (trajectories from an initial state to a given state that satisfy user-defined constraints) are estimated as a basis spanning the state space of the feasible solutions to the optimization objective. (iii) The next step is to compute a candidate trajectory that represents an optimal operation by combining trajectories generated in the previous step. (iv) In order to ensure the executability (fireability) of the candidate trajectory, post-filtering is executed by model checking (exhaustive simulation).

The contribution of the current paper compared to previous ones [11, 12] is the improved efficiency of the algorithm originating in (i) the basis-based generation of a candidate trajectory instead of using the Accelerated Branch and Bound algorithm, and (ii) a new algorithm to compute the reachability–membership function that does not require the explicit enumeration of the state space of the Petri net.

The paper is organized as follows. An introduction is given in Section 2 to Petri nets and Process Network Synthesis. The optimal trajectory problem is defined for Petri nets extended with cost parameters in Section 3. Section 4 recalls the assignments of the elements and problem semantics of the two paradigms followed by the analysis of the problems related to the direct algorithm adaptation and we recall the integrated technique in [11] to solve the Petri net-based optimization problem emphasizing the new methods for the generation of the candidate optimal trajectory in Section 5 and the reachability–membership function in Section 6. Section 7 provides an overview of related works, and finally, Section 8 reports on initial results and concludes our work.

2 Basic Notations

Hereafter only the notations and definitions of Petri nets and PNS problems necessary for understanding the concept of the solution (for more details see e.g., [20] and [7]) will be recalled.

2.1 Petri Nets

A *Petri net* is a directed, bipartite graph represented by a four-tuple $PN = \langle P, T, w, M_0 \rangle$, where P and T are the disjoint sets of place and transition nodes, respectively. Places may contain tokens, whose distribution represents the state of the net. The state of the net is described by the so-called marking. A marking is a $|P|$-dimensional vector over the naturals $\mathbb{N}$, where the i-th component $(M(p_i))$ is the number of tokens contained in $p_i \in P$. M_0 denotes the marking of the net in the initial state.

A Petri net with four places and three transitions is depicted on the left in Fig. 1 with the initial marking $M_0 = (2, 3, 0, 0)$.

The state of the net is changed by transition firings. The token flow is denoted by the weight function w assigning positive integers to the directed arcs between places and transitions $w : (P \times T) \cup (T \times P) \rightarrow \mathbb{N}$.

Let $\bullet x$ and $x\bullet$ denote the pre-set and the post-set of an element $x \in P \cup T$, respectively, s.t., $\bullet x = \{y : w(y, x) > 0\}$ and $x\bullet = \{y : w(x, y) > 0\}$.

A t transition is *enabled* at marking M (i.e. it may fire), if all the places in its pre-set contain at least as many tokens as required by the weight of the corresponding arc, i.e. if $\forall p \in \bullet t : w(p, t) \leq M(p)$ holds. The firing of the transition passes (removes and produces) the defined number of tokens (according to the weight function) from its input places $(\bullet t)$ to its output places $(t\bullet)$. Thus, the marking in the successor state M' can be estimated as $\forall p \in P : M'(p) = M(p) - w(p, t) + w(t, p)$.

Let us define the $|P| \times |T|$-dimensional *incidence matrix* W of the Petri net showing the change in the number of tokens in each place caused by the firing of the individual transitions, i.e. $W[p, t] = [-w(p, t) + w(t, p)], p \in P, t \in T$. Thus, $\forall p \in P : M'(p) = M(p) - w(p, t) + w(t, p) = M(p) + W(p, t)$, and $M' = M + W \cdot e_{t_j}, t_j = t$, where $e_{t_j} = (0, \ldots, 0, 1, 0, \ldots, 0)$ is a $|T|$-dimensional unit vector with 1 in its j-th component.

The effect of firing a transition can be extended for a firing sequence $s = \langle t_{i_1}, t_{i_2}, \ldots \rangle, 1 \leq i_k \leq |T|, k = 1, 2, \ldots$. The so-called *transition occurrence vector* (the Parikh vector of the sequence) of the firing sequence is a $|T|$-dimensional vector σ_s, where its jth component counts the number of firing transition t_j (i.e. $\sigma_s[j] = |t_{i_k} \in s : i_k = j|$. Then the firing sequence s starting from M_0 leads to marking M defined by the following so-called *state equation.*

$$M = M_0 + W \cdot \sigma_s. \tag{1}$$

Another mathematical paradigm, Process Network Synthesis was investigated in order to develop efficient search methods for the optimization of Petri net sequences with quantitative parameters.

2.2 Process Network Synthesis

Process Network Synthesis (PNS) algorithms were elaborated in chemical engineering to estimate an optimal resource allocation for the production of desired products from given raw materials.

A PNS problem is represented by the so-called *P-graph* [6]. A P-graph is a directed bipartite graph where the two sets of disjoint nodes are materials (or material stores) and operating units. An operating unit consumes its input materials (connected by incoming edges) in order to produce its output materials (connected by outgoing edges). Then the PNS problem is to produce all products in the required amount from available raw materials by the defined operating units of minimal cost.

The solution of the problem is a *sub-P-graph* or a *solution structure*. In order to ensure the production of the products a feasible network has to satisfy five axioms [8]: (A1) every final product is represented in the graph, (A2) a material has no input operating units if and only if it represents a raw material, (A3) every operating unit represents an operating unit defined in the synthesis problem, (A4) every operating unit has at least one path leading to a final product, and (A5) if a material belongs to the graph, it must be an input to or output from at least one operating unit in the graph.

In PNS problems, costs are assigned to raw materials and operating units (as fix and operating costs). Naturally, an *optimal network* for the PNS problem is a solution structure together with operating rates (assigned to the operating units) such that the production of the products is of a minimal cost. Based on the specific structure of the PNS problem, the combinatorial optimization problem can be solved efficiently by exploiting of the features given by the five axioms. The optimal solution structure of a PNS problems is synthesized by three algorithms [7].

The *Maximal Structure Generation (MSG)* and the *Solution Structure Generation (SSG)* algorithms collect the *structurally* feasible sub-P-graphs generating a topology spanning the state space for further optimization. The MSG algorithm generates the superstructure of the feasible solutions. MSG achieves a significant reduction of the solution space in polynomial time by excluding those materials and operating units from the initial P-graph that violate any of the five axioms.

Then the SSG algorithm computes the structurally feasible networks: it builds up a set of possible operating units recursively starting from the products. The algorithm maintains a set of materials *'to be produced'*. The iteration consists of two main steps: at first, a material from the set 'to be produced' is selected and excluded. Then the operating units producing the selected material are taken into consideration: a subset of them is added to the previously selected operating units and the algorithm calls itself recursively.

The SSG algorithm generates exactly the set of solution structures satisfying the five axioms as the spanning graph of the set of the selected operating units. The resulting solution structures formulate a logic basis for the optimization problem that is closed under unification: an arbitrary solution structure can be generated by this elementary solutions under the operations of unifications. This corresponds

to the fact, that if two recipes produce some material according to the five axioms, then their simultaneous operation is a proper solution, as well. Please observe, that the SSG algorithm neglects the quantitative parameters: the solution structures are structural solutions (given by the contained operating units).

PNS problems over linear constraints and objective functions can be represented as a mixed integer linear programming. The *Accelerated Branch and Bound (ABB)* algorithm [26] solves this programming problem by exploiting the additional structural properties (the same way as in the SSG algorithm): the algorithm delivers a structurally feasible network with the operating rates of the individual operating units.

In the ABB algorithm, the *numerical cut* (bounding) is conventional (for more details on Branch–and–Bound algorithms see [19]): if there is a known solution of a lower cost, a branch of a greater value is cut, and only branches with lower value are taken into consideration. In contrary to a conventional B&B algorithm, *logical cut* (branching) is based on the structural properties of the problem: branching is performed by the binary variables indicating whether a particular operating unit is involved in the candidate solution according to the building process of SSG. This way the ABB algorithm reduces the search for combinations to the elementary solutions (recipes) instead of performing trials with individual elementary operations (i.e. the fixing of binary variables in a certain order used in the conventional B&B techniques). Thus, an essential improvement can be achieved by algorithm ABB by the logical cuts in contrast to the conventional algorithm that traverses all the $2^{|T|}$ problem nodes in worst case.

In the following section we discuss how PNS algorithms can be adapted to the Petri net optimal trajectory problem.

3 Optimal Trajectory Problem

System optimization and verification, validation problems can be formulated as combined reachability and optimization problems: (i) it has to be decided, whether a particular state of the system is reachable from the initial state using the available resources, and (ii) if the state is reachable, an optimal trajectory has to be computed. For instance, let us take a resource allocation problem modeled by a Petri net. Program states can be modeled by a set of Petri net places, and resource allocation to tasks can be modeled by transitions. Hence, the problem of finding an optimal trajectory between the initial and the desired states can be translated into a so-called Petri net partial reachability problem.

A state (marking) is reachable from an initial state if there exists a firing sequence between the two markings such that the trajectory is compliant in each individual step with the firing condition. Frequently, in engineering problems only the marking of a subset of places is relevant from practical point of view. Therefore, 'partial reachability' means that we should reach a state M that *covers* the desired (partial) state $M_{partial}$, i.e. $M \geq M_{partial}$.

An optimal trajectory problem can be mapped into the search for a trajectory

between two Petri net states with the least/most cost, time, quality indicator, etc. As transitions typically represent operations in the system we restrict ourself to cost functions linear in the number of executed transitions (however, this approach can be adapted to other quantitative or even qualitative parameters).

A *Petri net with cost parameters* is a $PN_c = \langle PN, \vec{c} \rangle$, where the cost function $\vec{c} \in (\mathbb{R}^+ \cup \{0\})^{|T|}$ assigns costs to the firing of the individual transitions. Thus, *the cost of the firing sequence* can be interpreted as the sum of the cost values of the transitions in the sequence, formally, $c_s = \vec{c} \cdot \sigma_s$, where σ_s is the transition occurrence vector of the firing sequence s, and c_s denotes the cost of the firing sequence.

This way the optimal trajectory problem can be formulated as follows: a *Petri net optimal trajectory problem* $R_{PN} = \langle PN_c, M \rangle$ is a Petri net with cost parameters and a target marking M (i.e. a partial marking s.t., $M(p) = 0$ for each place $p \in P$ that is irrelevant) where the problem is to find a *fireable* (executable) trajectory s such that there exists a marking $M' \geq M$ that is reachable from M_0 by firing sequence s such that c_s is minimal. Then σ_s is called an *optimal transition occurrence vector.*

Example. An example Petri net is shown in Fig. 1 with cost parameters depicted as transition labels. The optimal trajectory problem shown is to find a trajectory of minimal cost from the given initial marking $M_0 = (2, 3, 0, 0)$ to a marking in which place p_4 contains at least one token.

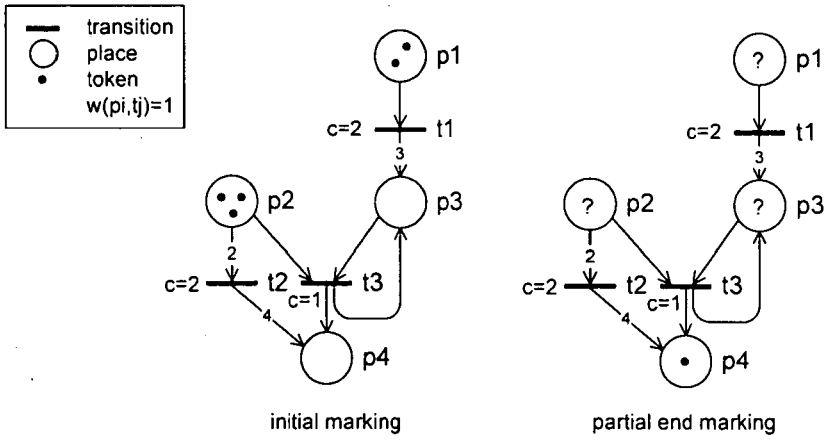


Figure 1: Example optimal trajectory problem

An obvious way to generate the optimal trajectory is the exhaustive traversal of the state space starting from the initial state. However, the exhaustive search for such a trajectory may easily result in state space explosion in case of complex systems. *In order to solve the Petri net reachability problem without state space explosion*, several techniques are discussed in the literature like unfolding-based

solutions for 1-bounded (safe) Petri nets [4], symbolic analysis using Binary Decision Diagrams (BDD) [21], or linear algebra–based algorithms [3, 24].

As an alternative solution, the state equation–based (Equation 1) method constitutes a mixed integer linear programming problem, as the components of the transition occurrence vector σ are nonnegative integers.

From the point of view of the reachability problem, the state equation method represents a *semi–decision* method as it formulates a necessary condition: if there is no σ that satisfies the inequality $M \leq M_0 + W \cdot \sigma$, then there is no firing sequence s from M_0 to a state that covers M. Otherwise, since the transition occurrence vector contains only reduced information on a trajectory by omitting the order of the individual firings, a solution vector σ of the state equation can represent a *spurious solution.* In this case, no appropriate firing sequence s exists such that $\sigma_s = \sigma$, despite the fulfillment of the state equation. Therefore the elimination of spurious solutions necessitates a separate check of the fireability of the solution transition vectors.

4 PNS Algorithms in the Solution of the Optimal Trajectory Problem

In order to support the modeling and solution of verification and optimization of real–time systems, we aimed at combining the PNS approach with the Petri net modeling in previous papers [12] and [11]. Although the two approaches are very similar, and PNS algorithms can be adapted to the optimal trajectory problem, the simple merging of ideas does not provide a proper solution in general.

4.1 The Optimal Trajectory Problem as a PNS Problem

As PNS algorithms focus only on the production of the desired end products without taking into account the intermediate byproducts, the PNS problem constitutes a partial reachability problem with cost parameters for the output places. Thus, a Petri net optimal trajectory problem *could* be described as a PNS problem, where (i) the places that are marked in the initial marking are the raw materials, (ii) the places that have to be marked at the target marking are the products, (iii) a transition is an operating unit mapping the pre- and post-set of the transition into the input and output materials of the operating unit, respectively (preserving the weight function), (iv) the cost of a transition is the operation cost of the corresponding operating unit, (v) and the result vector of the ABB algorithm can be interpreted as an optimal transition occurrence vector for the optimal trajectory problem.

The operating rates of operating units in chemical engineering may take continuous values, which is not allowed in case of a Petri net transition. Therefore the ABB algorithm cannot be directly used for the solution of the optimal trajectory problem but it can be reformulated so that the algorithm searches for integer

operating rates. The current paper proposes another method to deliver an integer solution based on the solution structures generated by the SSG algorithm (see Section 5).

On the other hand, thanks to their expressive power, Petri nets cover a wider range of models than P-graphs, i.e. not all Petri net reachability problems can be directly described as a PNS problem. One reason for the limited expressiveness of the PNS problems are the constraints introduced by the axioms (see Section 2.2) for the translated reachability problem. In [11] the problem of *'produced raw materials'* and *'catalysts'* is discussed.

In chemical engineering, *catalysts* play an essential role in production processes: they are both consumed and produced during the manufacturing process resulting in a total of zero change in their amount. However, while such catalysts cannot be assessed based upon material bill like equations as those used in the PNS model, PNS algorithms assume those materials to be available in the beginning.

In the Petri net optimal trajectory problem *catalysts* are tokens in places that participate in a cycle. In contrary to PNS problems, these catalysts may be required to the fireability of a sequence. Thus, it can occur, that there is no firing sequence in the solution structure of the PNS problem of the translated Petri net reachability problem. Thus, our problem has two different aspects requiring optimality, and fireability of the solution in order to be feasible.

4.2 Previous Results and New Contribution

In [11] we proposed an integrated algorithm. The algorithm consists of a collection of semi–decision methods that gradually eliminate the spurious non-fireable solutions as early as possible. The systematic search follows a best-first approach: the candidate solution nearest to the quantitative optimum is estimated as a transition occurrence vector and its feasibility (fireability) is checked until a solution is found. To provide a quick optimization technique for the reachability problem of general Petri nets, the optimization by the ABB algorithm (over integer variables) is complemented by a subsequent fireability check. This check is composed of a set of two subchecks: (i) semi–decisions like one use the *reachability–membership function* which is parameterized by two markings M_1, M_2 and evaluated to true if M_2 is reachable from M_1 or a fast *reachability check* of the marking reached by the solution transition occurrence vector from the initial marking (that is calculated by the state equation) eliminating a major part of spurious solutions, and (ii) an explicit *fireability check* of the transition occurrence vector providing the final evidence on the feasibility of solutions by using model checker SPIN [15].

Depending on the result of the checks, the algorithm either terminates delivering a fireable optimal solution or a new search is performed by the ABB algorithm for the second optimal solution. The main advantage of this approach is that both the reachability–membership function and the generated SPIN code can be reused to check the next candidate solution.

The new contribution in the current paper compared with previous works [11], [12] is (i) the generation of an integer candidate transition occurrence vector using a

Branch–and–Bound algorithm where branching is based on the *logical combinations of the basis solution structures* as variables (Section 5), and (ii) the introduction of an efficient reachability–membership function computation in the form of the *transitive closure of the single-step transition relation* (Section 6).

5 Solution Structure-based Optimization

In [11] we used the Accelerated Branch and Bound algorithm to generate the transition occurrence vector of an optimal candidate trajectory. However, the current ABB implementation potentially violates the required integrality of the Petri net reachability problem, i.e. it may return a non–integer as an optimal solution.

While the acceleration in the original ABB algorithm lies in the exploitation of the *correlation between the binary variables* assigned to the transitions (or to the operating units) our new Branch–and–Bound solution generates an optimal candidate trajectory for the Petri net optimal trajectory problem *based on the logic basis of the Petri net structure.* While the bounding step is identical in both approaches, instead of making decisions on the binary variables and their correlation at branching, our method considers *complete solution structures as a single boolean variable.*

The set of the logically feasible solution structures (satisfying Axioms (A1)-(A5) in Section 2.2) is closed under the logical unification. A basis of this set can be generated such that all solution structures can be composed as the logical union of some of these basis solution structures. This way our method covers all the feasible solutions.

Let be given an optimal trajectory problem $R_{PN} = (P, T, w, M_0, \vec{c}, M_p)$. At first, we generate a basis of the solution structures by a slightly modified version of the SSG algorithm. The modification lies in the collection step of a *basis* of the solution structures: a subset of the solution structures is called *solution structure basis* such that all other solution structures can be calculated as the disjunction (or union) of some of the basis solution structures. Since the set of the solution structures is closed under unification, this subset does exist and it can be computed. Thus, all new solution structure generated by the SSG algorithm has to be evaluated, whether it can be synthesized as the union of some previously generated solution structures or not.

The solution structure basis for the Petri net reachability problem constitute a set of characteristic vectors $S(PN)_{basis} = \{\theta_i\}$ representing the solution structures such that for all $j : 1 \leq j \leq |T|$:

$$\theta_i(j) = \begin{cases} 1 & \text{if } t_j \text{ is contained in the corresponding solution structure} \\ 0 & \text{otherwise.} \end{cases}$$

The candidate optimal transition occurrence vector σ_{PN_R} for the Petri net is an integer solution of the following mixed integer linear programming problem such

that σ_{PN_R} is generated as the *logical combination* (union) of the elements of the logic basis (i.e. the characteristic vectors of the solution structure basis).

$$\text{minimize } c \cdot \sigma_{PN_R}, \tag{2}$$

$$\text{subject to } M \leq M_0 + W \cdot \sigma_{PN_R}, \tag{3}$$

and the characteristic vector

$$\chi(\sigma_{PN_R}) \text{ of } \sigma_{PN_R} = \bigvee_{i=1}^{|S(PN)_{basis}|} \alpha(i) \cdot \theta_i \tag{4}$$

$$\sigma_{PN_R} \in \mathbb{N}^{|T|}, \alpha \in \mathbb{B}^{|S(PN)_{basis}|}, \mathbb{B} = \{0,1\} \tag{5}$$

The above problem can be solved by a Branch–and–Bound (B&B) algorithm. The starting problem consists of the objective function Eq. 2 and the constraint Eq. 3. Then the B&B tree is built by adding constraints due to Eq.4.

Eq.4 ensures that the logic structure of the candidate optimal transition occurrence vector is composed of some basis solution structures: the characteristic vector of the candidate solution corresponds to the logical disjunction of some basis solution structures. Numerically, there is a corresponding $\alpha \in \mathbb{B}^{|S(PN)_{basis}|}$ vector for all subproblem such that $\forall 1 \leq i \leq |T| : 0 = \sigma_{PN_R}(i) \iff \nexists 1 \leq k \leq |S(PN)_{basis}| : \alpha_k = 1 \wedge \theta_k(i) = 1$. In other words, exactly those transitions are included in the solution transition occurrence vector that are contained by some of the composing basis solution structures.

Branching. In the B&B algorithm branching is performed by the binary variables α_i considering the corresponding solution structure θ_i. The main difference between the ABB algorithm (Section 2.2) and our B&B algorithm is that the ABB algorithm decides on the binary variables according to the individual transitions while we search the solution over the logically feasible solution structures (i.e. over the set of the binary variables!).

Generation of the Integer Solution. since we need integer solutions, if we reach a leaf, i.e. the logical structure of the candidate solution is fixed, we solve the given restricted MILP problem.

Bounding and Pruning. Bounding is traditional: if the LP relaxation is infeasible or returns a worse cost value than a best known value, the tree is pruned.

An additional advantage of our algorithm is that the next best solution can be easily computed using the already built B&B tree.

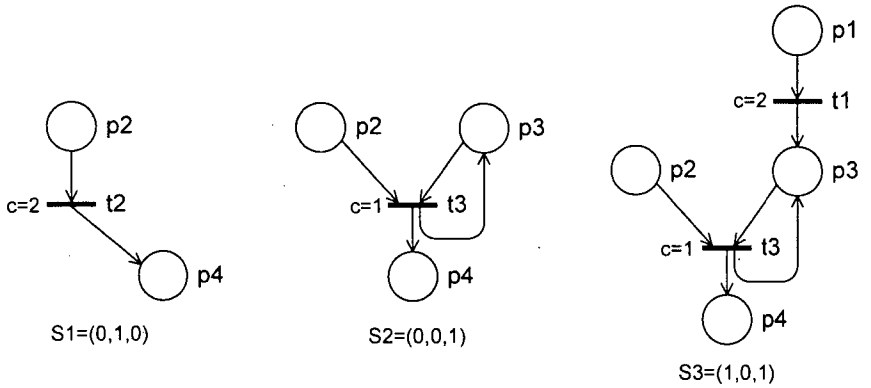


Figure 2: Solution structure basis for the reachability problem of the running example generated by the SSG algorithm

Example. Fig. 2 shows the solution structures in the solution structure basis for the reachability problem generated by the modified SSG algorithm. Please observe, that they are mutually independent under unification and there is only a single additional solution structure that can be composed as the logical union of some basis solution structures: the union of the three solution structures.

According to the example in Fig. 1, the candidate optimal transition occurrence vector σ_{PN_R} has to satisfy the following inequality system.

$$\text{minimize } (2,2,1) \cdot \sigma_{PN_R} \tag{6}$$

$$\text{subject to } (0,0,0,1) \leq (2,3,0,0) + \begin{pmatrix} -1 & 0 & 0 \\ 0 & -2 & -1 \\ 3 & 0 & 0 \\ 0 & 4 & 1 \end{pmatrix} \cdot \sigma_{PN_R}, \tag{7}$$

$$\chi(\sigma_{PN_R}) = \bigvee_{i=1}^{3} \alpha(i) \cdot \theta_i \tag{8}$$

$$\text{where } \sigma_{PN_R} \in \mathbb{N}^{|T|}, \alpha_i \in \mathbb{B} : \forall 1 \leq i \leq 3, \tag{9}$$

$$\text{and } \theta_1 = (0,1,0),\ \theta_2 = (0,0,1),\ \theta_3 = (1,0,1). \tag{10}$$

The Branch-and-Bound algorithm solved 7 LP relaxation, and 1 MILP problem, and the solution is $\sigma_{PN_R} = (0,0,1)$. The corresponding B&B tree is shown in Fig. 3.

Initially, the α_i variables are restricted neither to 1 nor to 0. The cost value and the solution σ_{PN_R} vector of the LP relaxation is shown below the constraints for α_i. Since the solution is non–integer valued, α_1 is restricted to 1 (see $Node_1$). Then we branch the tree taking α_2 into consideration. Since the unbounded α variables are between 0 and 1, m denotes a very small value in subproblems $Node_2$, $Node_3$, and

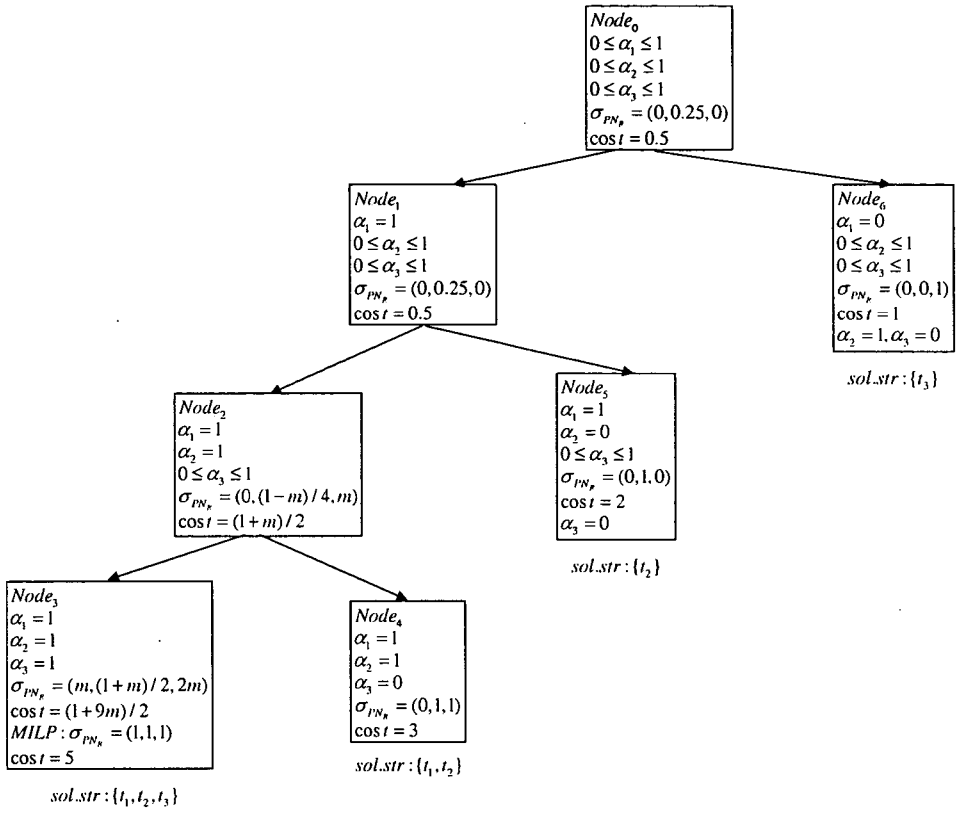


Figure 3: The B&B tree solving the optimal trajectory problem in Fig. 1

$Node_4$. In the leaf $Node_3$ the solution of the MILP problem returns the first best known solution $\sigma_{PN_R} = (1,1,1)$ with a cost of 5 units and the logical structure of the solution is composed of all the three solution structures.

At nodes $Node_4$, $Node_5$, ¡and $Node_6$ the solution of the corresponding LP relaxation returns integer solutions of a worse value thus, the tree can be pruned at these nodes. The candidate optimal solution is found at $Node_6$ composed of one basis solution structure θ_2. (Please note, that the depth-first algorithm required no bounding. However, if branching is done for instance in the order of $\alpha_2, \alpha_1, \alpha_3$, the optimal solution is found already at the first node.)

However, the solution ($S2$ in Fig. 2) represents a non-fireable solution from the initial marking $M_0 = (2,3,0,0)$, since transition t_3 is not fireable at marking $M_0 = (2,3,0,0)$ and no other transition is included in the solution. In order to eliminate these spurious solutions we introduce the following reachability and fireability checks.

6 Reachability–Membership Function Generation

We use symbolic modeling techniques introduced in [21] for the check of the candidate transition occurrence vector computed by the previous PNS algorithms-based programming problem. The efficiency of our new method lies in its parametral feature.

6.1 The Reachability–Membership Function Generation Algorithm

The states and the representation of the dynamic behavior (successor relation) of Petri nets are encoded as Boolean functions during the generation of the reachability–membership function (shortly reachability function). The algorithm generates the reachability function as the transitive closure of the single-step transition relation starting from the current set of reachable markings encoded as the disjunction of Boolean functions (collecting transitions that are enabled in the set of reachable markings). The computation of the set of new reachable states is performed by Boolean operations based on their *Binary Decision Diagram (BDD)* representation.

Due to the encoding of the reachable set by Boolean variables, the reachability function can be parameterized by the initial and the final states. The reachability function returns the value true, if the final state is reachable from the initial state and false, otherwise. The reachability function is characteristic for the Petri net, thus, it has to be estimated one by one, and *does not have to be newly generated* for the check of the next candidate solution after a spurious solution is found. The other advantage of the method compared to the previous one in [11] is that the states do not have to be explicitly enumerated as the parameters of the reachability function. Nevertheless, the main disadvantage of the method is its slightly limited scope (the Petri net has to be bounded) in order to be able to be parameterized and encoded by the function.

For the sake of simplicity, let $PN = \langle P, T, w, M_0 \rangle$ be a safe (or 1-bounded) Petri net, i.e. for every marking M reachable from the initial marking M_0, $\forall p \in P : M(p) \leq 1$ holds. Thus, the weight is 1 for each arc. (Note, that the algorithm also works for general bounded Petri nets using n Boolean variables for upper bound k s.t. $2^n \geq k$ [21], or using *Functional Decision Diagrams* ([25]).

A marking M is encoded as a set of Boolean variables $\{p_1, \ldots, p_n; n = |P|\}$ (as a characteristic function). Each variable p_i assigned to a place has a value of 1 if and only if the corresponding place is marked in M.

A set of markings can be represented by the disjunction (union) of their Boolean representations. The enabledness of a transition t in marking M is evaluated by the equation $E_t = \prod_{p_i \in \bullet t} p_i$. That means that the transition is enabled if there are at least as many tokens in its places $p \in \bullet t$ as required by the weight function that is 1 in our case. Since we supposed that the Petri net is a safe Petri net, this number of tokens is exactly 1, i.e. $p_i \equiv 1$.

The following transition function is used in order to compute the subsequent reachable states from the markings in the actual set. $\delta^t : P \rightarrow \{true, false\}$ is the *transition function* for $t \in T$, i.e. it denotes the effect of transition t on the Boolean variables after its firing.

$$\delta_i^t(p_1, \ldots, p_n) = \begin{cases} 1 & \text{if } p_i \in t\bullet \\ 0 & \text{if } p_i \in \bullet t \setminus t\bullet \\ p_i & \text{otherwise.} \end{cases}$$

The parameterized reachability function is generated iteratively as the *transitive closure of the single-step transition function.* Algorithm 1 sketches this stepwise generation of the reachability function. (Note, that the conjunction and disjunction operations on markings are interpreted as the corresponding operation on the place variables.)

The single-step transition function g is the disjunction of the conjunction of the transition function δ^t and the firing condition E_t for all transitions $t \in T$ (see line 1). The algorithm starts from the initial set of reachable markings, i.e. $f_0(v_1, \ldots, v_n, w_1, \ldots, w_n) = (v_1, \ldots, v_n \equiv w_1, \ldots, w_n)$ (see line 1). The actual reachability function $f_i(M_0, M)$ after the i-th iteration is evaluated to true if marking M is reachable from M_0 by a firing sequence with maximal length of i. Then the next reachability function f_{i+1} is calculated as the *disjunction of* the actual reachability function f_i and the single-step transition function g from the set of markings that are reachable in at most i steps from the initial marking (i.e. $f_i(M_0, M_1) \wedge g(M_1, M)$, see line 1. Both the single-step transition function and the reachability function are parameterized by variables $(v_1, \ldots, v_n)$, $(r_1, \ldots, r_n)$, $(w_1, \ldots, w_n)$ denoting the initial, the auxiliary and the end markings, respectively. The algorithm terminates if the computation reached a fixpoint, i.e. the transitive closure is computed (at line 1).

Since the generation of the reachability function does not store the order of the transition firings, the described method is appropriate only to check the reachability of the marking computed by the state equation. On the other hand, if we were interested only in the reachability of this end marking and we would not like to reuse the reachability function, we could improve our method in the following way.

- The sum of the components of the candidate transition occurrence vector provides the necessary number of iterations. Thus, after exceeding this number of iterations, the computation of the reachability function can be stopped.
- The transition function could be restricted only to those transitions that are involved in the candidate transition occurrence vector.

Example. In the running example (see in Fig. 1), transition t_1 is enabled if there is a token in p_1. Thus, the enabledness of transition t_1 can be expressed as follows: $E_{t_1} = p_1 = (p_1 \equiv 1)$. The firing of transition t_1 removes the token from p_1 and produces one token to place p_3, while the marking of places p_2 and p_4 do not change. Namely, the transition function for t_1 is $\delta^{t_1}(p_1, p_2, p_3, p_4) = (0, p_2, 1, p_4)$.

Algorithm 1 Reachability–membership function generation

1: Let be a safe Petri net $PN = \langle P, T, w, M_0 \rangle$ given.
2: Compute the ROBDD of the transition function g:
3: $g(v_1, \ldots, v_n, w_1, \ldots, w_n) = \bigvee_{\forall t \in T} \left[\bigwedge_{i=1}^{n} (w_i \equiv \delta_i^t(v_i)) \cdot E_t \right]$
4: $\{g(v_1, \ldots, v_n, w_1, \ldots, w_n) = 1 \iff \exists t \in T : \forall i : 1 \leq i \leq n : \delta_i^t(v_i) \equiv w_i$ and $E_t \wedge v_1 \ldots v_n = 1.\}$
5: Compute the ROBDD of the 0-th iteration of the reachability function:
6: $f_0(v_1, \ldots, v_n, w_1, \ldots, w_n) = (v_1 \ldots v_n \equiv w_1 \ldots w_n) = \bigwedge_{i=1}^{n} (v_i \equiv w_i)$
7: $i \leftarrow 0$
8: **repeat**
9: Compute the ROBDD of f_{i+1} using the ROBDDs of f_i and g:
10: $f_{i+1}(v_1, \ldots, v_n, w_1, \ldots, w_n) = f_i(v_1, \ldots, v_n, w_1, \ldots, w_n) \vee f_i(v_1, \ldots, v_n, r_1, \ldots, r_n) \wedge g(r_1, \ldots, r_n, w_1, \ldots, w_n)$
11: {the reachability function $f_i(M_0, M)$ is evaluated to true if and only if M is reachable from M_0 by a firing sequence with maximal length of i.}
12: $i \leftarrow i + 1$
13: **until** $f_i \equiv f_{i+1}$.

According to the above algorithm, in our running example functions f_0, g, and the reachability function f are the following. Lines 12-14 in the definition of function g refer to the effect and the enabledness of transition t_1, t_2, and t_3, respectively. The reachability function f reflects the firing sequences on a parameterized marking: the 'sub-equation' in line 16 is evaluated to true only for end markings $M = (w_1, w_2, w_3, w_4)$ that are the same as the initial marking $M_0 = (v_1, v_2, v_3, v_4)$ while the Boolean expressions in lines 17-19 and 20-21 are evaluated for markings reachable by a firing sequence of length 1, and 2, respectively.

$$f_0(v_1v_2v_3v_4, w_1w_2w_3w_4) = (w_1w_2w_3w_4 \equiv v_1v_2v_3v_4) \tag{11}$$
$$g(v_1v_2v_3v_4, w_1w_2w_3w_4) = (w_1w_2w_3w_4 \equiv 0v_21v_4) \wedge (v_1 \equiv 1) \vee \tag{12}$$
$$(w_1w_2w_3w_4 \equiv v_10v_31) \wedge (v_2 \equiv 1) \vee \tag{13}$$
$$(w_1w_2w_3w_4 \equiv v_1011) \wedge (v_2 \equiv 1) \wedge (v_3 \equiv 1) \tag{14}$$

$$f(v_1v_2v_3v_4, w_1w_2w_3w_4) = \tag{15}$$
$$(w_1w_2w_3w_4 \equiv v_1v_2v_3v_4) \vee \tag{16}$$
$$(w_1w_2w_3w_4 \equiv 0v_21v_4) \wedge (v_1 \equiv 1) \vee \tag{17}$$
$$(w_1w_2w_3w_4 \equiv v_10v_31) \wedge (v_2 \equiv 1) \vee \tag{18}$$
$$(w_1w_2w_3w_4 \equiv v_1011) \wedge (v_2 \equiv 1) \wedge (v_3 \equiv 1) \vee \tag{19}$$
$$(w_1w_2w_3w_4 \equiv 0011) \wedge (v_1 \equiv 1) \wedge (v_2 \equiv 1) \vee \tag{20}$$
$$(w_1w_2w_3w_4 \equiv 0011) \wedge (v_1 \equiv 1) \wedge (v_2 \equiv 1) \wedge (v_3 \equiv 1) \tag{21}$$

6.2 Implementation of the Reachability Function: Binary Decision Diagrams

In order to gain efficiency, the reachability function computation requires an efficient representation of Boolean functions. As Binary Decision Diagrams (BDDs) [2] provide an efficient form to manipulate Boolean functions, we express the above Boolean functions by means of them. BDDs are directed acyclic graphs with two leaf nodes that represent Boolean functions 0 and 1. The other nodes represent the Boolean variables and each of them has two outgoing edges labeled by 1 'then' or 0 'else'. Thus, the evaluation of a Boolean function is performed by the traversal of the corresponding BDD according to the actual values of the variables.

To generate the reachability function (see Algorithm 1) as a BDD, we use Reduced Ordered BDDs (ROBDD) where the equivalent branches of the tree are merged and the redundant variables are excluded from the tree. Then the disjunction and conjunction operations in the calculation of g, f_0, and f_{i+1} (in lines 1, 1, 1) can be directly executed on the ROBDD representations. However, the ROBDDs to be merged have to contain the same variables in the same order. Nevertheless, arbitrary variables can be added to the ROBDD of functions f_i and g thus, the computation of f_{i+1} is performed by using the ROBDDs of f_i and g with variables $v_1, \ldots, v_n, r_1, \ldots, r_n, w_1, \ldots, w_n$. Since variables $r_1, \ldots, r_n$ in g can be substituted by variables $v_1, \ldots, v_n$, the resulting ROBDD of f_{i+1} contains only variables $v_1, \ldots, v_n, w_1, \ldots, w_n$ satisfying the definition of f such that it is interpreted on the variables of the initial and the end markings.

7 Related work

The use of integer programming methods in the analysis of Petri nets is not a novel idea. In [18] *deadlock detection* was reduced to a mixed integer linear programming problem. In [17] the authors presented a further development of this approach to prove *deadlock detection, mutual exclusion, and marking reachability and coverability.* In contrary to our approach, this solution was based on the *unfolding* of the Petri net. Another unfolding-based solution is discussed for *safe* Petri nets in [4].

Linear algebraic algorithms were used to solve the Petri net reachability problem without state space explosion in [3, 24]. Although these techniques are powerful, in general they provide only semi-decision techniques to decide the reachability of a given marking while we are dealing with general Petri nets.

Several papers [14, 9] use *stochastic Petri nets* or *timed Petri nets* to model and solve scheduling and optimization problems (e.g. in the field of manufacturing systems). These approaches use mainly simulation and performance evaluation in order to solve the problem (for instance, in [28] profit function values are represented as a function according to some given restrictions using simulation of stochastic Petri nets). Since we do not only want to solve the optimal trajectory problem (where we have fixed parameters assigned to the transitions), but we also aim at *simultaneous verification and optimization*, these techniques are not appropriate

for our purposes.

Model checking methods were used to solve scheduling and optimization problems in several papers. These papers are common in the feature that the problem is translated into a reachability condition that can be easily encoded into a temporal logic expression to be verified by the model checking tool. In [5] timed automata was used to model a steel plant and the corresponding scheduling problem. The scheduling problem was solved using the *UPPAAL* tool, which is a model checker for networks of timed automata. [23] and [13] deal with time optimization problems using the SPIN model checker. The main idea of this solution (originated from [23]) is to interpret Branch-and-Bound techniques encoding both the bounding and branching conditions into the linear temporal expression to be verified. Although the optimal trajectory problem could be solved using only SPIN based on the above technique the pre–optimization in our method using linear programming solutions significantly reduces the search space for the optimal solution.

In [16] the authors analyzed the *executability* of a given process network solution,where operating units consume exactly one unit of their input materials and produce exactly one unit of their output materials. This modified PNS problem was solved using an *automaton theoretical approach* such that the problem is transformed into a problem to find the shortest path in the weighted transition graph of the automaton constructed from the PNS problem. In case of Petri nets where the weights of the arcs are restricted to one, the fireability of the candidate solution transition occurrence vector can be proved using this method.

8 Conclusion and Initial Results

Finally, our method was tested on an example Petri net with 11 places and 7 transitions. The example originates in [1]: we reformulated the PNS problem as a Petri net optimal trajectory problem. The example Petri net is shown in Figure 4, where the weight of the arcs are written on the edges.

The original SSG algorithm generates 19 solution structures (see the solution structures in [1]) while our modified algorithm computes only 7 basis solution structures. The Branch–and–Bound algorithm (described in Section 5) executes 13 LP relaxations and 5 MILP problem (at the leaves of the tree). This number of LP relaxations and MILP problem solving is still less than the number of the MILP problems to be solved if we would like to find the optimal solution calculating the optimal solution for all the 19 solution structures. The SSG basis-based algorithm returns the solution $\sigma_{PN_R} = (0, 1, 0, 0, 1, 0, 1)$ with cost 1085, i.e. the candidate optimal transition occurrence vector contains transitions $t2$, $t5$, and $t7$ (each exactly once).

As a next step, we calculate the end marking by the state equation. Then we check whether there exists a trajectory from the given initial marking to the calculated end marking using the reachability function. Substituting the initial and the end marking into the reachability function, the answer is *positive*, i.e. the end marking is reachable from the initial marking. In case of a negative answer, the

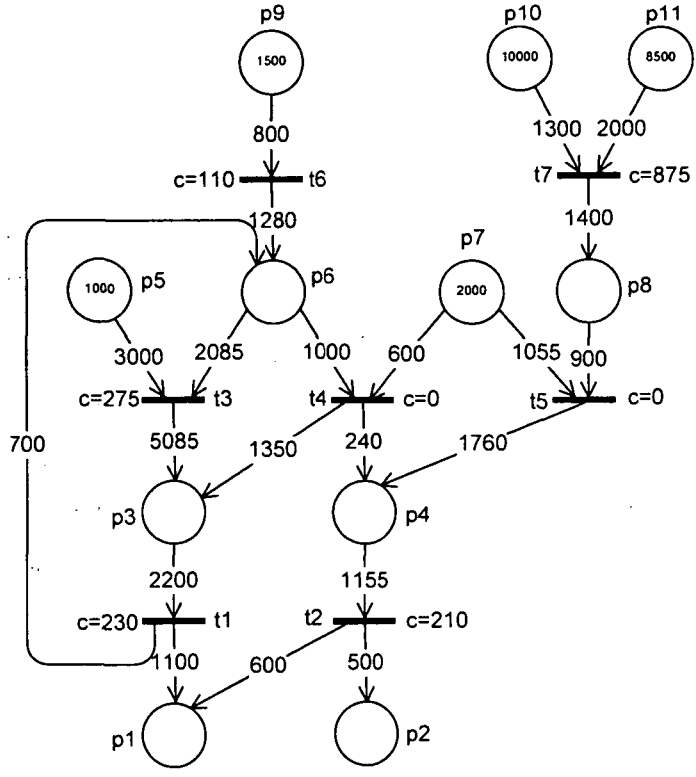


Figure 4: Example optimal trajectory problem

next optimal solution should be generated by our Branch–and–Bound algorithm for which the previous calculations could be reused, as well as the reachability function for the next check.

Finally, we generate the fireable trajectory according to the candidate optimal transition occurrence vector, that is the following transition sequence: $\langle t7, t5, t2 \rangle$.

Since the solution of the optimal trajectory problem needs the generation of the fireable trajectory, the efficiency of our method cannot be easily compared to other approaches. In addition, several structural specialties could be explored, for which dedicated algorithms give much more efficient results. In Table 1 the number of LPs and MILPs solved are shown both in case of ABB (however it does not return an integer solution!!!) and in case of our method, while the execution time values (on a Pentium IV) for the individual phases of the algorithm are given in Table 2.

Conclusion In the current paper we introduced a solution structure-based method for optimization purposes to solve the Petri net optimal trajectory problem and the generation of a reusable, initial marking–independent reachability function

Table 1: Initial results: comparison of number of LPs and MILPs solved by the ABB algorithm and the solution structure basis-based B&B algorithm

algorithm	number of LPs	number of MILPs
ABB		0
B&B algorithm based on the solution structure basis	13	5

Table 2: Initial results: execution time for the example optimal trajectory problem

step	execution time (sec)
Generation of the solution structure basis $S(PN)_{basis}$	0.01
Generation of the optimal candidate transition occurrence vector σ_{PN_R}	0.72
Generation of the reachability function	4.56
Trajectory generation using SPIN	0.01

to check the reachability of the target state corresponding to the candidate optimal solution structure. As further work, we aim to develop a similar framework for time optimization with Petri nets using dedicated linear programming problem solutions developed for the so-called S-graphs [22].

References

[1] M. H. Brendel, F. Friedler, and L. T. Fan. Decision–Mapping: A Tool for Consistent and Complete Decisions in Process Synthesis. *Computers Chemical Engineering*, 24:1859–1864, 2000.

[2] R. Bryant. Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[3] J. Desel. *Petrinetze, Lineare Algebra und lineare Programmierung*, volume 26 of *Teubner-Texte zur Informatik*. B. G. Teubner Stuttgart-Leipzig, 1998.

[4] J. Esparza and C. Schröter. Unfolding Based Algorithms for the Reachability Problem. *Fundamenta Informaticae*, 46:1–17, 2001.

[5] A. Fehnker. Scheduling a Steel Plant with Timed Automata. In *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA '99)*. IEEE Computer Society Press, 1999.

[6] F. Friedler, L. T. Fan, and B. Imreh. Process Network Synthesis: Problem Definition. *Networks*, 28(2):119–124, 1998.

[7] F. Friedler, K. Tarjan, Y. W. Huang, and L. Fan. Combinatorial Algorithms for Process Synthesis. *Computers Chemical Engineering*, 16:313–320, 1992.

[8] F. Friedler, K. Tarjan, Y. W. Huang, and L. Fan. Graph-Theoretic Approach to Process Synthesis: Axioms and Theorems. *Chemical Engineering Science*, 47(8):1973–1988, 1992.

[9] T. Gu, P. A. Bahri, and G. Cai. Timed Petri-net based formulation and an algorithm for the optimal scheduling of batch plants. *International Journal of Applied Mathematics and Computer Science*, 13(4):527–536, 2003.

[10] S. Gyapay. Model Transformation from General Resource Models to Petri Nets using Graph Transformation. Technical report, Technical University of Berlin, Dept. of Computer Science, July, 2004. ISSN 1436-9915.

[11] S. Gyapay and A. Pataricza. A combination of Petri nets and Process Network Synthesis. In *2003 IEEE International Conference on Systems, Man & Cybernetics, Invited Sessions/Track on "Petri Nets and Discrete Event Systems"*, pages 1167–1174, Washington, D.C., USA, October 5-8 2003.

[12] S. Gyapay, A. Pataricza, J. Sziray, and F. Friedler. Petri Net-based optimization of production systems. In A. Lovrenčić and I. J. Rudas, editors, 6^{th} *IEEE International Conference on Intelligent Engineering Systems*, pages 465–469. Organized and published by Faculty of Organization end Informatics, University of Zagreb, Croatia, May 26–28 2002.

[13] S. Gyapay, A. Schmidt, and D. Varró. Joint Optimization and Reachability Analysis in Graph Transformation Systems with Time. *Electronic Notes in Theoretical Computer Science*, 109:137–147, 2004.

[14] I. Hatono, K. Yamagata, and H. Tamura. Modeling and Online Scheduling of Flexible Manufacturing Systems Using Stochastic Petri Nets. *IEEE Trans. Softw. Eng.*, 17(2):126–132, 1991.

[15] G. J. Holzmann. *The SPIN Model Checker - Primer and Reference Manual.* Addison-Wesley, Boston, USA, 2003.

[16] B. Imreh. Automaton theory approach for solving modified PNS Problems. *Acta Cybernetica*, 15(3):327–338, 2002.

[17] V. Khomenko and M. Koutny. Verification of Bounded Petri Nets Using Integer Programming. Technical report, Department of Computing Science, University of Newcastle upon Tyne, 2000. Technical Report CS-TR-711.

[18] S. Melzer and S. Römer. Deadlock Checking Using Net Unfoldings. In *CAV '97: Proceedings of the 9th International Conference on Computer Aided Verification*, pages 352–363. Springer-Verlag, 1997.

[19] M. Minoux. *Mathematical Programming: Theory and Algorithms.* John Wiley & Sons, 1986.

[20] T. Murata. Petri nets: Properties, analysis and applications. In *Proc. IEEE*, volume 77, pages 541–580, 1989.

[21] E. Pastor, J. Cortadella, and O. Roig. Symbolic Analysis of Bounded Petri Nets. *IEEE Transactions on Computers*, 50(5):432–448, 2001.

[22] J. Romero, L. Puigjaner, T. Holczinger, and F. Friedler. Scheduling Intermediate Storage Multipurpose Batch Plants Using the S-Graph. *AIChE Journal*, 50(2):403–417, 2004.

[23] T. C. Ruys. Optimal scheduling using Branch-and-Bound with SPIN 4.0. In *Proc. 10th International SPIN Workshop*, volume 2648 of *LNCS*, pages 1–17, Portland, Oregon, USA, May 9–10 2003. Springer.

[24] M. Silva, E. Teurel, and J. M. Colom. Linear algebraic and linear programming techniques for the analysis of Place/Transition net systems. In G. R. W. Reisig, editor, *Lectures on Petri Nets I:Basic Models*, volume 1791 of *LNCS*, pages 309–373. Springer, 1998.

[25] R. S. Stankovic. Functional Decision Diagrams for Multiple-Valued Functions. In *25th IEEE International Symposium on Multiple-Valued Logic*, pages 284–189, 1995.

[26] J. B. Vajda, F. Friedler, and L. Fan. Parallelization of the Accelerated Branch and Bound Algorithm of Process Synthesis: Application in Total Flowsheet Synthesis. *Acta Chimica Slovenica*, 42(1):15–20, 1995.

[27] D. Varró and A. Pataricza. VPM: Mathematics of metamodeling is metamodeling mathematics. *Journal of Software and Systems Modelling*, 3(2):187–210, 2003.

[28] A. Zimmermann, D. Rodríguez, and M. Silva. Modelling and Optimization of Manufacturing Systems: Petri Nets and Simulated Annealing. In *Proceedings of the 1999 European Control Conference ECC99*, Karlsruhe, Germany, August 1999.

Acta Cybernetica 17 (2005) 247–271.

Coordination Language for Distributed Clean*

Zoltán Horváth†, Zoltán Hernyák‡, and Viktória Zsók†

Abstract

The distributed evaluation of functional programs and the communication between computational nodes require high-level process description and coordination mechanism. This paper presents the D-Clean high-level functional language, which supports the distributed computation of Clean functions over a cluster. The lazy functional programming language Clean is extended by new language elements in order to achieve parallel features. The distributed computations of functions are expressed in the form of process-networks. D-Clean introduces language primitives to control the dataflow in a distributed process-network.

A process scheme defines a partial computation graph, where the nodes are functions to be evaluated and the edges are communication channels. The computational nodes are implemented as statically typed Clean programs. The schemes are parameterized by functions, types and data for defining process networks.

D-Clean is compiled to an intermediate level language called D-Box. The D-Clean generic constructs are instantiated into D-Box expressions. D-Box is designed for the description of the computational nodes. D-Box expressions hide implementation details and enable direct control over the process-network. The asynchronous communication is based on language-independent middleware services.

The present paper provides the syntax and the informal semantics of both coordination languages. To illustrate the definition of a distributed functional computational pattern using the D-Clean language a farm skeleton running example is presented.

Keywords: D.1 Programming Techniques: D.1.1 Applicative (Functional) Programming, D.1.3 Concurrent Programming. Distributed functional programming, coordination language, Clean, functional skeleton.

*Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr.T037742 and by the Bolyai Research Scholarship.

†Department of Programming Languages and Compilers, Eötvös Loránd University, Hungary. e-mail: hz@inf.elte.hu, zsv@inf.elte.hu

‡Department of Information Technology, Eszterházy Károly College, e-mail: aroan@ektf.hu

Introduction

Nowadays it is prevailing to develop and to test parallel functional applications on PC clusters [13]. The distributed evaluation of functional programs and the communication between computational nodes require high-level process description and coordination mechanism [1, 15]. Therefore it became important to provide distributed environments making possible the development of applications with client-programs written in functional programming languages. The proposed D-Clean language is an extension of the functional programming language Clean and supports the distributed computation of Clean functions. The computation of functions is expressed in the form of distributed process-networks. D-Clean primitives control the dataflow in the process-networks.

Skeletons are computation patterns, algorithmic schemes that captures common computation mechanism. Skeletons [12, 17] can be defined and parameterized by functions, types and data. They are widely used in parallel computations. In functional programming skeletons can be combined with evaluation strategies in order to obtain optimal parallel behaviour [11]. D-Clean supports the composition of the coordination primitives to build compound coordination structures. The coordination structures are used to define distributed functional computational skeletons, process schemes. D-Clean schemes are parameterized by types and by functions. Before instantiation the actual values of the type parameters have to be inferred from the type description of the embedded Clean expressions[1]. In the case of the widely used skeletons (like farm, divide and conquer, pipe and reduce [4, 5, 8]) it is easier to deal with the type inference problem than in general.

D-Clean is compiled to an intermediate level language called D-Box, similar to the idea of [2]. D-Box is designed for the description of the computational nodes implemented as Clean programs, which use middleware services for asynchronous communication [9, 20]. D-Box expressions hide implementation details and enable direct control over the process-network.

The D-Clean control language has a higher level coordination role, while D-Box has a lower abstraction level. The syntax and informal semantics of both coordination languages are described. We also present a mapping from D-Clean expressions to D-Box expressions.

The D-Box language is a description language for the source codes of computational nodes. In this language input and output protocols can be defined. A transformation of D-Box definitions into Clean language programs is described. A graphical developer environment was built to support a direct use of the D-Box language.

In section 1 the main concepts of the D-Clean language are presented. As an example a farm computation is defined. Section 2 presents the context-free syntax of the D-Clean language. The semantics of the D-Clean language is described in section 3 in an informal way. In section 4 the D-Box expressions corresponding to the farm example are included. Section 5 defines the context-free syntax of the

[1]Similar to the C++ template parameter deduction [19].

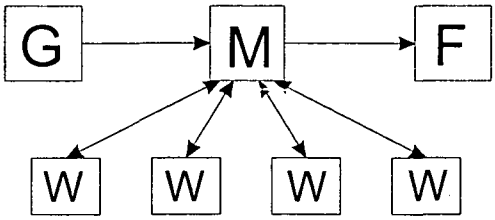


Figure 1: Farm scheme

D-Box language. The semantics of the D-Box language is described in section 6 also in an informal way and in section 7 the mapping from D-Clean to the D-Box language is given. Section 8 shows an example how to compile D-Box definitions into Clean language programs using channels. Section 9 presents measurements about the object code to be generated. A second example is presented in section 9 to demonstrate the expressiveness and the ease-of-use of D-Clean and the D-Box graphical developer environment. In section 10 the related works are discussed and in the section 11 the conclusion closes the paper.

1 An example written in D-Clean

The present paper uses the well-known farm skeleton as running example (see Figure 1). The farm skeleton divides the input data into n parts. The n parts of the original input are sent to n different worker processes. Each worker applies the same function on the partial data and calculates the n sub-results. The final result can be combined when every worker completes the tasks.

In our example the farm scheme is used as a simple distributed computation, where the computation node 'G' generates a finite list of integer numbers to be sorted. The master node 'M' receives the sortable data, splits into sublists and sends them to its workers (a worker is denoted by 'W'). The workers sort the sublists and send them back to the farm master node ('M'). The master node receives the sublists and merges them on-the-fly using the comparer function `lessThan`. The finally produced sorted list can be forwarded to the last computational node (denoted by 'F') for any further processing.

A D-Clean program consists of a start expression, in which a collection of user-defined D-Clean process schemes can be applied. A process scheme itself is written in D-Clean too. The start expression is given as the `DistrStart` function definition.

D-Clean coordination structures are mappings between communication channels and are designed as generic templates parameterized by types and by functions. The value of type parameters are determined by type inference. The templates are instantiated by the D-Clean pre-compiler at compile time.

The matching of types between the base types of channels and the types of embedded Clean expressions is a static semantic requirement. A D-Clean expression may be a compound expression or a direct use of coordination primitives. Process

scheme definitions are named D-Clean expressions with formal parameters. A process scheme library can be built using the coordination primitives and the already defined schemes[2].

In our running example a user-defined scheme FARM can be applied in the start expression in the following way:

```
DistrStart =
   (DStop finish) (FARM comb solve divide N) (DStart generator)
      where
         generator = [1,4,2,3,8,6]
         finish    = (WriteResultDat "sorted.dat")
         comb      = (mergeSort lessThan)
         solve     = qsort
         divide    = (divide N)
         N         = 2
```

The above start expression is a *composition* of three coordination structures. It applies two D-Clean language coordination primitives (`DStop` and `DStart`) and the user-defined scheme called FARM (see below the D-Clean definition of the FARM scheme).

By `Ch a` we denote the type of a channel carrying elements of type `a`. A D-Clean expression is a mapping from a sequence of input channels to a sequence of output channels. The sequence of the types of the input and output channels of a coordination structure is given in the `<Ch a, Ch b, ...>` form. In our example the inferred signatures of the instantiated control structures are as follows:

```
DStart generator ::  <Ch Int>
FARM :: <Ch Int> -> <Ch Int>
DStop finish ::  <Ch Int>
```

`DStart` generates the data to be sorted (corresponds to the box 'G' in Fig. 1). An expression called `generator` is used for generating the input data. The `generator` is actually a Clean function with the output type `[Int]`. The generated data are sent via a channel to the FARM coordination structure, which computes the sorted list and forwards it to the last component of the computation. Finally `DStop` applies the `WriteResult "sorted.dat"` function on the sorted list received from the farm in order to save the result (it corresponds to the box 'F' in Figure 1). The pseudo codes of `DStart` and `DStop` primitives are presented in the appendix.

The definition of the process scheme FARM uses four parameter functions: `comb`, `solve`, `divide` and `n`. The process scheme is the composition of the coordination primitives `DMerge`, `DApply` and `DDivideS`:

```
SCHEME FARM comb solve divide n =
   (DMerge comb) (DApply solve) (DDivideS divide n)
```

[2] We present a solution to the problem of the recursively called instances of the coordination structures in a language extension of the D-Clean [10].

The process scheme FARM composes the three actions taken by the farm computational pattern: first the incoming data list is divided into n parts using the parameter function `divide` as the parameter of `DDivideS`. After that the `solve` function is applied by `DApply` on every sub-list. Finally their sub-results are collected and merged by `DMerge` into one list applying the parameter function `comb`.

The instance of the process scheme FARM has the signature `<Ch Int> -> <Ch Int>`, while the instances of the components have the following signatures[3]:

```
DDivideS divide n :: <Ch Int> -> <Ch Int, Ch Int, ..., Ch Int>
DApply solve :: <Ch Int, ..., Ch Int > -> <Ch Int, ..., Ch Int>
DMerge comb :: <Ch Int, Ch Int, ..., Ch Int> -> <Ch Int>
```

The set of the worker boxes corresponds to the boxes generated from the `DApply solve` expression.

The functions `mergeSort lessThan`, `qsort`, `divide`, `N` are the actual parameters of the FARM process scheme in our example. The role of the master node is shared between two computation nodes, implementing the tasks of `DDivideS` and `DMerge` respectively. The input is divided into `N` pieces by the user defined `divide` function, where `N=2` is a constant value. The sublists are sent to the worker nodes using a special `splitk` output protocol (see section 3). The workers solve the main task - the sorting of the sublists using the `qsort` standard Clean function. After sorting the sublists, the workers send their results to the collector node, which receives the input sublists and merges them using the `mergeSort lessThan` function.

2 The syntax definition of the D-Clean language

We introduce the following extensions to the standard BNF syntax:

- **{notion}+** means that the notion occurs at least once,
- **{notion}*** means that the notion occurs zero, one or more times,
- **{notion}-list** means one or more occurrences of the notion separated by commas,
- **terminals** are closed between apostrophes.

⟨DISTART_RULE⟩ :=="DistrStart" "=" ⟨DEXPR⟩
⟨DEXPR⟩ :=⟨DPRIMITIVE⟩ | ⟨SCHEME_NAME⟩ { ⟨act_param⟩ }*
| ⟨DEXPR⟩ ⟨DEXPR⟩
⟨DPRIMITIVE⟩ :=⟨DStart_USE⟩ | ⟨DStop_USE⟩ | ⟨DMap_USE⟩ |
⟨DDivideS_USE⟩ | ⟨DMerge_USE⟩
⟨SCHEME_DEF⟩ :=="SCHEME" ⟨SCHEME_NAME⟩ { ⟨formal_param⟩}*
"=" { ⟨DEXPR⟩ }+
⟨SCHEME_NAME⟩ :={⟨UpCaseLetter⟩ }+

[3]The description of the type inference system in general is out of the scope of this paper.

Every D-Clean program contains exactly one start expression given as the right-hand side of the `DistrStart` definition. A scheme is a compound D-Clean coordination structure parameterized by types and by functions. The actual parameters of schemes are Clean expressions [16]. The identity function is the simplest Clean expression which can be embedded into a D-Clean expression.

The type of the arguments of the Clean expressions determines the type of the communication channels, which is restricted by the limitations of the used middleware interface. Due to these limitations we say that $\mathcal{T}$ is a transmissible type, if $\mathcal{T}$ is `Int`, `Real`, `Bool`, `Char` or a record built from these basic Clean types. At this time functions cannot be transferred through channels.

⟨act_param⟩ :== ⟨fun_expr⟩
⟨fun_expr⟩ :== ⟨clean_expr⟩ | "(|" ⟨DEXPR⟩ "|)" | ⟨fun_expr⟩ ⟨fun_expr⟩
⟨formal_param⟩ :== ⟨identifier⟩

The direct use of coordination primitives is a parameterized form of basic dataflow structures.

⟨DStart_USE⟩ :== "DStart" ⟨act_param⟩
⟨DStop_USE⟩ :== "DStop" ⟨act_param⟩
⟨DDivideS_USE⟩ :== "DDivideS" ⟨act_param⟩ ⟨number⟩
⟨DMerge_USE⟩ :== "DMerge" ⟨act_param⟩
⟨DMap_USE⟩ :== ⟨Simple_DMap_DEF⟩ | ⟨Multi_DMap_DEF⟩
⟨Simple_DMap_USE⟩ :== ⟨DApplyVariations⟩ ⟨act_param⟩
⟨Multi_DMap_USE⟩ :== ⟨DApplyVariations⟩ "[" { ⟨act_param⟩ }-list "]"
⟨DApplyVariations⟩ :== "DApply" | "DMap" | "DReduce" | "DProduce"
| "DFilter"
⟨UpCaseLetter⟩ :== "A" | "B" | "C" | "D" | ... | "Z"

3 Informal semantics of the D-Clean language

This section presents the newly introduced coordination primitives in an informal way. The figures of the section illustrate the working mechanism. F denotes the function expression embedded into the coordination primitive.

The coordination structures use channels for receiving the input data required for the arguments of their function expressions. The results (components of a k tuple in general case) of the function expression are sent to the output channels. Every channel is capable of carrying data elements of a specified base type from one computational node to another one. We use the unary algebraic type constructor **Ch a** to construct channel types, where the base type **a** is a transmissible type.

A coordination primitive usually has two parameters: a function expression (or a list of function expressions) and a sequence of input channels. The coordination primitives return a sequence of output channels. The signature of the coordination primitive, i.e. the types of the input and output channels are inferred according

to the type of the embedded Clean expressions. In the following the `aCh` denotes a channel type, while `aCh*` denotes a finite sequence of channel types.

DStart fun_expr :: aCh*

The task of `DStart` primitive is to start the distributed computation by producing the input data for the dataflow graph. It has no input channels, only output channels. The results of the `fun_expr` are sent to the output channels. Each D-Clean program contains at least one `DStart` primitive (see Figure 2).

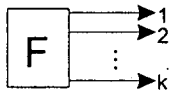


Figure 2: DStart node

DStop fun_expr :: aCh* -> <>

When a function expression embedded into a `DStop` primitive has k arguments, then the computation node evaluating the expression needs k input channels. Each input channel carries one argument for the function expression.

The task of this primitive is to receive and save the result of the computation. It has as many input channels as the function expression requires, but it has no output channels. `DStop` closes the computational process. Each D-Clean program contains at least one `DStop` primitive (see Figure 3).

`DStop` is the last element of the D-Clean composition, the last element of the control flow. In some cases when the control flow contains forks, the network has multiple `DStop` elements.

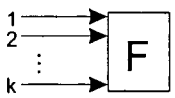


Figure 3: DStop node

DApply fun_expr :: aCh* -> aCh*

This variant of `DApply` applies the same function expression n times (see Figure 4/a) on $n * k$ channels. When the function expression has k arguments of types: $t_1, t_2, ..., t_k$, the number of input channels is $n * k$. The types of the arguments periodically match the type of the channels:
$< Ch\ t_1, Ch\ t_2, ..., Ch\ t_k, Ch\ t_1, Ch\ t_2, ..., Ch\ t_k, ..., Ch\ t_1, Ch\ t_2, ..., Ch\ t_k >$.
If the expression produces a tuple with m elements of the type $(p_1, p_2, ..., p_m)$, then

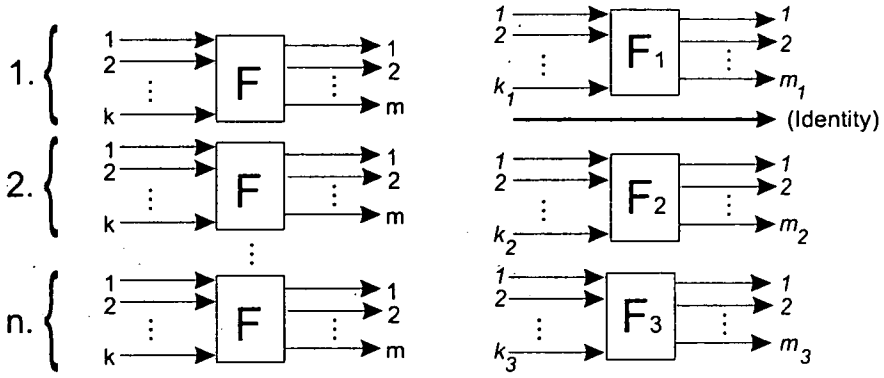


Figure 4: DApply variant a) and variant b).

the output channel sequence will contain $m * n$ elements, repeating the m type-sequences n times:
$< Ch\ p_1, ..., Ch\ p_m, Ch\ p_1, ..., Ch\ p_m, ..., Ch\ p_1, ..., Ch\ p_m >$.

DApply <fun_expr> :: aCh* -> aCh*

The second variant of `DApply` may apply different function expressions, which are given in the `<fun_expr>` sequence. The types and the number of the arguments of the function expressions can also be different. If the `<fun_expr>` sequence contains an identity function, then data received via the corresponding channel is directly forwarded to the next node.

The sequence of the input channels is constructed out of the channels required by the function expressions in the `<fun_expr>` sequence. The output sequence of channels is built up according to the results obtained by applying the function expressions. For example `DApply <`F_1`, id, `F_2`, `F_3`]>` yields the structure presented in Figure 4/b.

DFilter (a -> Bool) :: aCh* -> aCh*
DFilter <a -> Bool> :: aCh* -> aCh*

The `DFilter` primitive filters the elements of the input channels using a boolean function. It has two variants similarly to `DApply`. This is the D-Clean variant of the standard *filter* library function. This variant filters the incoming data elements before sending them to the outgoing channels.

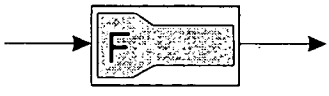


Figure 5: A DFilter node

DMap fun_expr :: aCh* -> aCh*
DMap <fun_exp> :: aCh* -> aCh*

DMap is a special case of DApply where the function expression must be an elementwise processable function [11]. It is the D-Clean variant of the standard *map* library function. It modifies the incoming data elements processing them one by one.

A valid parameter function expression for DMap can be a function expression either of type a->b or of type [a]->[b]. Suppose we have a list of n sublists as input data, then the qsort::[!a]->[a] sorting function[4] is a valid function expression as parameter for DMap. It takes every sublist element of the input list and applies the parameter function expression, i.e. the qsort function on it. The result will be the list of the n sorted sub-lists.

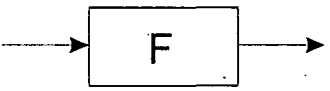


Figure 6: DMap nodes

DReduce fun_expr :: aCh* -> aCh*
DReduce <fun_expr> :: aCh* -> aCh*

DReduce is another special case of DApply with similar restrictions. A valid expression for DReduce has to decrease the dimension of the input channel type[5]. A valid expression has the type of form [a]->b. For example the sum::[a]->a function - which computes the sum of the elements of the input list - is a valid expression for DReduce.

DProduce fun_expr :: aCh* -> aCh*
DProduce <fun_expr> :: aCh* -> aCh*

DProduce is another special case of DApply. The expression has to increase the dimension of the channel type[6]. A valid expression must be of the form a->[b]. For example the divisors::Int->[Int] function - which generates all the divisors for an integer number - is a valid expression for a DProduce.

DDivideS fun_expr n :: aCh* -> aCh*

DDivideS is a static divider (see Figure 7). The expression splits the input data list into n parts and broadcasts them to n computational nodes. This primitive is called static divider since the value of n must be known at pre-compile time.

[4]! denotes strict evaluation of the argument.
[5]For example: list of lists → list.
[6]For example: list → list of lists.

The base type of the sublists has to be the same type as of the original list. Therefore the types of the output channels are the same as of the input ones. Consequently there will be n output channels.

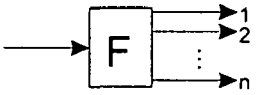


Figure 7: DDivideS node

DMerge fun_expr :: `aCh*` -> `aCh*`

`DMerge` collects the input sublists from channels and builds up the output data lists. All the input channels must have the same type (see Figure 8).

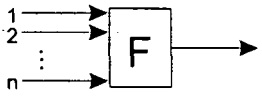


Figure 8: DMerge node

DLinear <fun_expr> :: `aCh*` -> `aCh*`

`DLinear` is a special coordination primitive. It simplifies the definition of the pipeline computation graph, where the nodes are connected to each other in a linear way (see Figure 9).

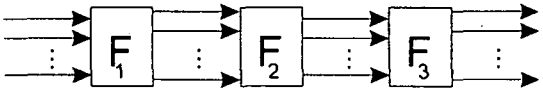


Figure 9: DLinear nodes

`DLinear` <$expr_1, expr_2, \ldots, expr_k$> is equivalent to the following composition of `DMap` primitives: `(DMap` $expr_k$`)` ... `(DMap` $expr_2$`)` `(DMap` $expr_1$`)`.

4 Examples on D-Box language

The D-Box language is used to generate the Clean code for a computational node. D-Clean expressions are mapped to D-Box definitions, the details of the mapping are given in section 7. Every box definition describes a computational node which contains an embedded expression, the input protocol and the output protocol (see Figure 10).

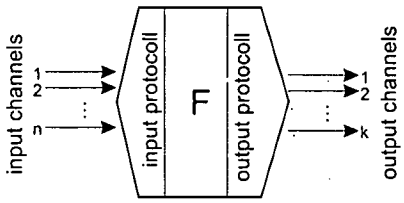


Figure 10: A computational node

```
BOX <BOXID>
  { { <INPUT_DEF> }, { <EXPRESSION_DEF> }, { <OUTPUT_DEF> } }
```

A computational node may use more than one input channel. At this level a channel identification mechanism is used. One input channel is described by its type and by the unique id of the channel. Notation $[\mathcal{T}]$ is used in the type description of a channel, which is used to transfer a single list of elements of the base type $\mathcal{T}$. Whenever a list of lists is sent via a channel, type $[[\mathcal{T}]]$ is associated to it.

The input protocol also determines the synchronization mode of the input channels. There are three modes: `memory`, `joinl` and `joink` (see section 5). The input is completely defined when the list of the input channels (`<INPUT_CHANNEL_LIST>`) and the input protocol (`INPUT_PROC_MODE`) are given.

The number and/or the base types of the input channels can be different from the types of arguments of the expression (`<ARGUMENT_TYPE_LIST>`). The matching of channel types to argument types is completed at code generation time according to the actual protocol. The same holds for the `<RESULT_TYPE_LIST>` too.

The output protocol definition has the same structure as the input definition. A complete D-Box definition has the following parts:

```
BOX <BOXID>
{  { (<INPUT_CHANNEL_LIST>), INPUT_PROC_MODE },
   { (<ARGUMENT_TYPE_LIST>), <EXPRESSION>, (<RESULT_TYPE_LIST>) },
   { (<OUTPUT_CHANNEL_LIST>), OUTPUT_PROC_MODE } }
```

The running example presented in section 1 is mapped into D-Box expressions. A detailed description of the generated D-Boxes is given in the following.

The `BoxID_00` definition describes a computational node, which generates the data. Because it requires no input, there are no input channels, and the input protocol is the *memory* protocol. It produces a list of integers, which values are sent to the channel with the id #1. The *split1* protocol means a one-to-one mapping of the components of the results to the output channels.

```
BOX BoxID_00  // for DStart generator
{    { ( null ), memory }, // INP CHNS and PROT
     { ( null ); generator, ( [Int] ) }, // EXPR
     { ( ( [Int], 1 ) ), split1 } // OUTP CHNS and PROT }
```

The `BoxID_01` definition describes the first task of the farm master node (the node marked with 'M' in the farm scheme in Figure 1). It receives integer elements

from channel #1. The *join1* protocol reads this input channel and passes the data elements to the expression as arguments. The expression applies the `divide` function on it with the constant parameter `N`. The result is a list of N sublists. In the running example $N = 2$. These two sub-lists are sent to channels #2 and #3 by the *splitk* output protocol.

```
BOX BoxID_01  // for DDivideS divide N
{     { ( ( [Int], 1 ) ),  join1 }, // INP CHNS and PROT
      { ( [Int] ), divide N, ( [[Int]] ) }, // EXPR
      { ( ( [Int], 2 ), ( [Int], 3 ) ), splitk } // OUTPUT }
```

The `BoxID_02` definition implements the first farm worker node. It receives the input list from channel #2, then sorts the list using the `qsort` function. The sorted list is sent to channel #4.

The *split1* protocol sends the elements of the result directly to a channel.

```
BOX BoxID_02   // for DMap qsort
{      { ( ( [Int], 2 ) ), join1  },
       { ( [!Int] ), qsort, ( [Int] ) },
       { ( ( [Int], 4 ) ), split1 } }
```

The `BoxID_03` describes the second farm worker node. The only differences are the id-s of the input and output channels: #3 and #5 respectively.

The `BoxID_04` definition presents the D-Box code of the second job of the farm master node. The data received from the two farm worker nodes on channels #4 and #5 are merged. After reconstructing the list of lists it applies the `mergeSort lessThan` function composition. In this particular case the expression merges two sorted lists and sends to channel #6.

The *joink* protocol merges the different input channels and constructs a list of lists. The elements of sub-lists are received on different channels.

```
BOX BoxID_04 // for DMerge (mergeSort lessThan)
{    { ( [Int], 4 ), ( [Int], 5 ) ), joink },
     { ( [[Int]] ), mergeSort lessThan, ( [Int] ) },
     { ( ( [Int], 6 ), split1 }}
```

The last D-Box definition describes the final box. It receives the input data from channel #6 and saves it to a file. The box has no output channel, so the *memory* protocol is used.

```
BOX BoxID_05 // for DStop (WriteResultDat "sorted.dat")
{    { ( ( [Int], 6 ), join1 },
     { ( [Int] ), WriteResultDat "sorted.dat",  (null ) },
     { ( null ), memory } }
```

5 Syntax definition of the D-Box language

The D-Box language has a lower abstraction level for describing the distributed computation. Each D-Box definition defines one computational node. The definition consists of three parts: the input protocol, the embedded expression and the

output protocol. Both protocols contain the descriptions of the channels and the processing mode.

⟨BOXDEF⟩ :== "BOX" ⟨BoxID⟩
"{" ⟨InpProt⟩ "," ⟨ExpressionDef⟩ "," ⟨OutProt⟩ "}"

The expression part contains the specification of the types of the arguments, the Clean expression itself and the types of the components of the result.

⟨ExpressionDef⟩ :== "{" "(" ({⟨TypeDef⟩}-list | "null") ")" "," ⟨Expression⟩ ","
"(" ({⟨TypeDef⟩}-list | "null") ")" "}"

The expression can be a pure Clean expression, or a composition of Clean functions and embedded D-Clean expressions. An embedded D-Clean expression must be lifted out [10]. It generates a sub-graph, which has an entry and an exit box. On this level (D-Box level) the box id-s of these boxes must be given as arguments of a *BOXES* expression.

⟨Expression⟩ :== ⟨BoxesExpr⟩ | ⟨CleanFv⟩ | ⟨Expression⟩ ⟨Expression⟩
⟨BoxesExpr⟩ :== "BOXES" "(" ⟨BoxID⟩ "," ⟨BoxID⟩ ")"
⟨BoxID⟩ :== ⟨string⟩

The input section contains the description of the channels (types and id-s) and the concept of synchronizing and mapping the incoming data elements.

⟨InpProt⟩ :== "{" ({⟨IChannelDef⟩}-list|"null") "," ⟨InpProtMain⟩ "}"
⟨InpProtMain⟩ :== "join1" | "joink" | "memory"
⟨IChannelDef⟩ :== "(" ⟨TypeDef⟩ "," ⟨IChannelID⟩ ")"
⟨IChannelID⟩ :== ⟨Number⟩

Similarly the output section defines the output channels and the output protocol.

⟨OutProt⟩ :== "{" {⟨OChannelDef⟩}-list ";" ⟨OutProtMain⟩ "}"
⟨OutProtMain⟩ :== "split1" | "splitk" | "memory"
⟨OChannelDef⟩ :== "(" ⟨TypeDef⟩ "," ⟨OChannelID⟩ ")"
⟨OChannelID⟩ :== ⟨Number⟩

Transmissible types, and list or list of list of transmissible types are allowed.

⟨TypeDef⟩ :== ⟨TypeName⟩ | "[" ⟨TypeName⟩ "]" | "[[" ⟨TypeName⟩ "]]"

6 Informal semantics of the D-Box language

A box defines a computational node which is a Clean language program. It receives input data from input channels, then executes the computation and sends the results to output channels. The channels in our environment are remote objects providing operations to retrieve data elements from and to store elements. A computational node may perform the following actions:

1. connects to the input channels and gathers their identifiers into a list,
2. reads the input data elements from all the input channels,
3. processes the input data elements using the embedded expression of the box,
4. calculates the result of the computation,
5. connects to the output channels and gathers their identifiers into a list,
6. sends the result to the channels.

Some of these actions may overlap in time as a consequence of the lazy evaluation strategy of the host functional language.

The protocols define the processing mode of the input and the output channels, including the mapping of the input channels to the arguments of the expression. First we enumerate the protocol names, then we define their meanings.

The input protocols are the following: `memory`, `join1`, `joink`, while the output protocols are: `memory`, `split1`, `splitk`. In the following we give a detailed description of the above mentioned actions and protocols.

If a box has no input channels, then actions 1 and 2 are not performed. The input protocol is the `memory` protocol.

When the input arguments of the expression are carried by different input channels, a one-to-one mapping between arguments and channels is required. This input protocol is `join1`.

Let us consider an example. The first channel (list_a) is processed lazily, but the second one is processed strictly (list_b). The difference is given by the type definition of the embedded expression. The ! annotation indicates the strict evaluation mode, since the default evaluation is the lazy one. Observe that the lazy or strict processing of the channels depends on the type definition of the expression. The processing mode is automatically implemented by the protocol.

```
expr:: [a] [!b] -> [c]
list_a = list of all the elements from input channel 1
list_b = list of all the elements from input channel 2
result  = expr list_a list_b
send result to the output channel
```

Let us observe how a strictness annotation controls the semantics of the protocol. The first parameter of the `expr` is a list of `a` evaluated lazily. Reading and evaluating of the next element of `list_a` will always be postponed until it is really needed by the evaluation of the expression. The type of the second argument is annotated for a strict evaluation. The Clean reduction system is forced to read and evaluate the whole `list_b` before the evaluation of the expression starts.

An argument of the expression may have the type of list of lists (sub-lists). When elements belonging to different sub-lists are carried by different input channels, then the list of lists is built by the input protocol `joink`.

Similar rules are valid for the output protocols. When the result of the expression is a tuple $(r_1, r_2, \ldots, r_k)$, we need k output channels, one for each result component. In this case the output protocol is the `split1` protocol. The result

elements are sent to the output channels one by one. When a computation is terminated (all the input elements are processed), then a terminal signal is sent to all the output channels.

When the expression produces list of sub-lists, then the `splitk` protocol may be used. In this case we need as many output channels as many sub-lists the result list contains. The `splitk` protocol sends the sub-lists to different channels. When the box needs no output channel (for example a DStop-box), the output protocol is `memory`.

7 Mapping from D-Clean to D-Box

First we define several functions. The `elementTypeOf` function gives the base type of the given list. Type inference is done at compile time. It is required to determine all the types of the necessary channels before the code generation starts.

The `lengthOf` function determines how many elements are in a finite list. In case a list is a list of lists, then `lengthOf` determines the number of the sublists. The `nextChannelID` function generates the next free channel id number which was never used before and they are positive integer numbers. The `nextBoxID_nn` function generates a unique box id. It is required to be in form "BoxID_nn" and must be unique.

Here we give the structure of a TUP, the base type used in the description of the coordination structures at D-Box level:

TUP = (TypeDef, Id)

where `TypeDef` defines the type of the channel and the `Id` is a unique identification number of the communication channel.

In addition we define an `OUTPROT_LIST` expression using the following algorithm. The algorithm processes the type of the components of the result of a given expression and generates the output protocol list for the box. The output protocol list is constructed according to the result types of the expression and generating id-s for the channels in parallel.

```
OUTPROT_LIST :: [TypeDef] -> [TUP]
OUTPROT_LIST []     = []
OUTPROT_LIST [x:xs] = [(x, nextChannelID) : OUTPROT_LIST xs]
```

Function `INPROT_LIST` processes a list of types (the list of the types of the arguments of an expression) and a TUP list in parallel and generates the input protocol list for the box. The types of the arguments has to match the types of the channels.

```
INPROT_LIST :: [TypeDef] [TUP] -> [TUP]
INPROT_LIST [] [] = []
INPROT_LIST [x:xs] [(t, c) : ts]
   | match(x,t)    = [(t, c) : INPROT_LIST xs ts]
   | otherwise     = abort "Error!"
```

For each D-Clean coordination primitives we give the description of the generated boxes including the protocols used in the following.

The input protocol of a `DStart` structure is `null`. The output channel list is defined by processing the output types of the expression. This primitive uses the `split1` protocol. The result of the mapping is a box definition and a TUP list of output channels.

$\mathcal{D}$([**DStart expr**] = [(B, RESULT_OUTPUT_TUP_LIST)], where

```
RESULT_OUTPUT_TUP_LIST = OUTPROT_LIST (outputTypeOf expr)
B = BOX NextBoxID_nn
{   { (null), memory },
    { inputTypeOf expr, expr, outputTypeOf expr }
    { RESULT_OUTPUT_TUP_LIST, split1 } }
```

The output protocol of a `DStop` primitive is always `null`. The input channel list is determined by the previous control structure in the computational graph. The result of the mapping is a box definition and an empty output TUP list.

$\mathcal{D}$ [**DStop expr TUP_list**] = [(B, RESULT_OUTPUT_TUP_LIST)], where

```
RESULT_OUTPUT_TUP_LIST = []
BOX NextBoxID_nn
{   { INPROT_LIST (inputTypeOf expr) TUP_List, join1 },
    { inputTypeOf expr, expr, outputTypeOf expr },
    { (null), memory } }
```

The `DApply` primitive is mapped to n box definitions (see Section 3). One box uses k channels as its own input channels from the TUP list. The mapping is done when all the channels are bound. Each box will contain the same expression. The input protocol is always `join1` and the output protocol is `split1`. The result output TUP list is the merged list of all the output channels of all the boxes. The result also contains a set of boxes.

$\mathcal{D}$ [**DApply expr TUP_list**] = F TUP_list [] expr, where

```
F [] Y expr = Y
F TUP_List Y expr =
  F (drop k TUP_List) (Y ++ [(B, OUTPUT_TUP_LIST)]) expr
 where
  OUTPUT_TUP_LIST = OUTPROT_LIST (outputTypeOf expr)
  k = lengthOf (inputTypeOf expr)
  B = BOX NextBoxID_nn
  { { INPROT_LIST (inputTypeOf expr) (take k TUP_list), join1 },
    { inputTypeOf expr, expr, outputTypeOf expr },
    { OUTPUT_TUP_LIST, split1 } }
```

The second variant of `DApply` is mapped to more box expressions (see Section 3), but each box contains different expressions. Each box uses different numbers of channels as their input channels according to the actual number of the arguments. If the expression is the identity expression, there is no need to define a real box. The result output TUP list is the merged list of all the output channels of all the boxes. The result also contains a set of boxes.

$\mathcal{D}$ [**DApply expr_list TUP_list**] = `F TUP_LIST expr_list []`, where

```
F TUP_list [] Y = Y
F TUP_List [expr:xs] Y
    | expr == id
        = F (drop k TUP_List) xs (Y ++ [(B, OUTPUT_TUP_LIST)])
    | otherwise
        = F (drop k TUP_list) xs [Y : (B, OUTPUT_TUP_LIST)]
  where
    OUTPUT_TUP_LIST = OUTPROT_LIST (outputTypeOf expr)
    k = lengthOf (inputTypeOf expr)
    B = BOX NextBoxID_nn
    { { INPROT_LIST (inputTypeOf expr) (take k TUP_list), join1 },
      { inputTypeOf expr, expr, outputTypeOf expr },
      { OUTPUT_TUP_LIST, split1 } }
```

The following keywords are special cases of the `DApply` coordination primitive and their semantics can be given in an analogous way:

$\mathcal{D}$ [**DMap expr**] = $\mathcal{D}$[`DApply expr`]
$\mathcal{D}$ [**DMap expr_list**] = $\mathcal{D}$ [`DApply expr_list`]
$\mathcal{D}$ [**DReduce expr**] = $\mathcal{D}$ [`DApply expr`]
$\mathcal{D}$ [**DReduce expr_list**] = $\mathcal{D}$ [`DApply expr_list`]
$\mathcal{D}$ [**DProduce expr_list**] = $\mathcal{D}$ [`DApply expr`]
$\mathcal{D}$ [**DProduce expr_list**] = $\mathcal{D}$ [`DApply expr_list`]
$\mathcal{D}$ [**DFilter expr**] = $\mathcal{D}$ [`DApply (filter expr)`]
$\mathcal{D}$ [**DFilter** $[expr_1, expr_2, \ldots, expr_k]$] =
$\mathcal{D}$ [`DApply` $[filter\ expr_1, filter\ expr_2, \ldots, filter\ expr_k]$]

`DLinear` keyword is a special case of the `DMap` primitive, so the semantics of it can be given as the following composition:

$\mathcal{D}$ [**DLinear** $[expr_1, expr_2, \ldots, expr_n]$] =
$\mathcal{D}$ [`DMap` $expr_n$ ... `DMap` $expr_2$ `DMap` $expr_1$]

`DDivideS` is mapped to a box definition where the output protocol is always `splitk`. The output TUP list contains N elements. The divider expression splits the input list into N sub-lists.

$\mathcal{D}$ [**DDivideS expr N TUP_list**]= `[ (B, RESULT_OUTPUT_TUP_LIST) ]`, where

```
RESULT_OUTPUT_TUP_LIST = OUTPROT (outputTypeOf expr) N
OUTPROT _ 0 = []
OUTPROT l k = OUTPROT_LIST l ++ OUTPROT l (k-1)
B = BOX NextBoxID_nn
    { { INPROT_LIST (inputTypeOf expr), TUP_list, join1 },
      { inputTypeOf expr, expr, outputTypeOf expr },
      { RESULT_OUTPUT_TUP_LIST, splitk } }
```

`DMerge` is mapped to a box definition where the input protocol is always `joink` and the output protocol is `split1`. The result of the mapping produces an output TUP list.

$\mathcal{D}$ [DMerge expr TUP_list] = [(B, RESULT_OUTPUT_TUP_LIST)], where

```
RESULT_OUTPUT_TUP_LIST = OUTPROT_LIST (outputTypeOf expr)
B = BOX NextBoxID_nn
{   { INPROT (inputTypeOf expr), TUP_List, joink },
    { inputTypeOf expr, expr, outputTypeOf expr },
    { RESULT_OUTPUT_TUP_LIST, split1 } }
where
    INPROT _ [] = []
    INPROT l t  = INPROT_LIST l t ++ INPROT l (drop k t)
    k = lengthOf (InputTypeOf expr)
```

The problem of the embedded D-Clean expressions is discussed in [10].

8 Mapping from D-Box to Clean

The conversion between a D-Box code and a Clean program is straightforward. We can use several functions defined in the middleware interface library. The library contains skeletons for the pre-compiler, who instantiates them according to the actual input types. First we initialize the middleware. Each computational node communicates with at least one other computational node. Notation # introduces a let expression, while #! forces an immediate evaluation of the let expression. #! induces a sequential evaluation of the expressions. Function CHANNEL_FIND is implemented according to the actual middleware.

The code generated from the following D-Box definition is presented:

```
BOX BoxID_04  // for DMerge (mergeSort lessThan)
{     { ( [Int], 4 ), ( [Int], 5 ) ), joink },
      { ( [[Int]] ), mergeSort lessThan, ( [Int] ) },
      { ( ( [Int], 6 ), split1 } }
```

After the middleware is initialized, the input channel list is built up by using their ChannelID-s.

```
Start w =
   #! w                 = MIDDLEWARE_INIT w
   # (inp_chan_1,w)     = CHANNEL_FIND 4 w
   # (inp_chan_2,w)     = CHANNEL_FIND 5 w
   # input_channels     = [inp_chan_1, inp_chan_2]
```

ChannelID-s are determined from the input protocol section of the box. Afterwards, the input process is started in order to gather all the incoming data elements from the input channels.

```
   # (input_data,w)     = Joink input_channels w
```

Then we process the data:

```
   # result             = mergeSort lessThan input_data
```

Before sending the result, connections to the output channels are required. The ChannelID-s included into the box output protocol section are used.

```
# (outp_chan_1,w) = CHANNEL_FIND 6 w
# output_channels = [outp_chan_1]
```

The resulted data are sent to the output channels:

```
# w               = Split1 output_channels result w
```

The complete generated code of our D-Box example is given here:

```
Start w =
   #! w              = MIDDLEWARE_INIT w
   # (inp_chan_1,w)  = CHANNEL_FIND 4 w
   # (inp_chan_2,w)  = CHANNEL_FIND 5 w
   # input_channels  = [inp_chan_1, inp_chan_2]
   # (input_data,w)  = Joink input_channels w
   # result          = mergeSort lessThan input_data
   # (outp_chan_1,w) = CHANNEL_FIND 6 w
   # output_channels = [outp_chan_1]
   # w               = Split1 output_channels result w
   = w
```

9 Measurements

A sequential and several parallel implementations of the running example are measured and compared. We sorted integer lists of 250, 500, 1000, 2000, 4000 elements and we used 2 and 4 PC-s for the parallel implementations. The first diagram (see Figure 11) shows the computation time in seconds (Y axis) and the length of the lists (X axis). We used a weighted comparer function `lessThan` to slow down the computation simulating a complicated comparison of two data elements.

The diagram of the Figure 12 shows the speed-up. The diagrams show that

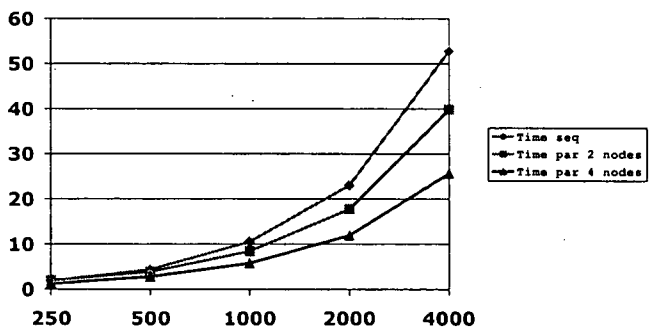


Figure 11: The computation time of the FARM skeleton

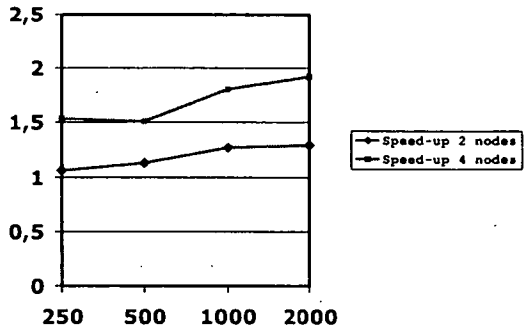


Figure 12: The speed-up of the FARM skeleton

the distributed implementation is efficient using the mapping from D-Clean to D-Box. A slight increasing of the speed-up can be observed. In case the workers are computing a weighted function, the overheads obtained during the communications are negligible.

Similar measurements can be found in earlier papers [9, 20].

Henceforth a second example is presented to *demonstrate the expressiveness* and ease-of-use of D-Clean. A matrix m is given with k columns and k rows. A sequence of vectors $vs =< v_1, \ldots v_n >$ is generated, the size of each vector is k. The sequence of products $< v_1 * m, \ldots, v_t, \ldots, v_n * m >$ has to be calculated elementwise. We compute the sequence sequentially, but the k elements of each product $(v_t * m)(1..k)$ is calculated in a parallel way. Every vector v_t is replicated in k copies $(v_{t_1}, \ldots v_{t_k})$ first and the scalar products of the columns and a copy of the vector, $(\sum_{j=1}^{k}(v_{t_i}(j) * m(j)(i)))$ is computed in parallel for every $i \in [1..k]$.

We present the structure of a D-Clean solution and the diagram of the corresponding process network. We omit the details of the D-Box definitions, which may be obtained by compiling D-Clean to D-Box or by generating D-Box source text by the D-Box graphical developer tool [6] (see Figure 13).

```
DistrStart =
    (DStop saver) (DMerge vectorize)
    (DApply [multiply column_1, multiply column_2, ..., multiply column_k])
    (DDivideS repeater k) (DStart vector_generator)
  where
    k = 4
    saver = saveToFile "result.dat"
    vectorize = id
    column_1 = getColumn matrix 1
    ...
    column_k = getColumn matrix k
    repeater k inp = take k (repeat inp)  // k copies
vector_generator:: [[Int]] // generates the vectors
getColumn:: [[Int]] Int -> [Int] // gets the ith column
multiply::[Int] [Int] -> Int
```

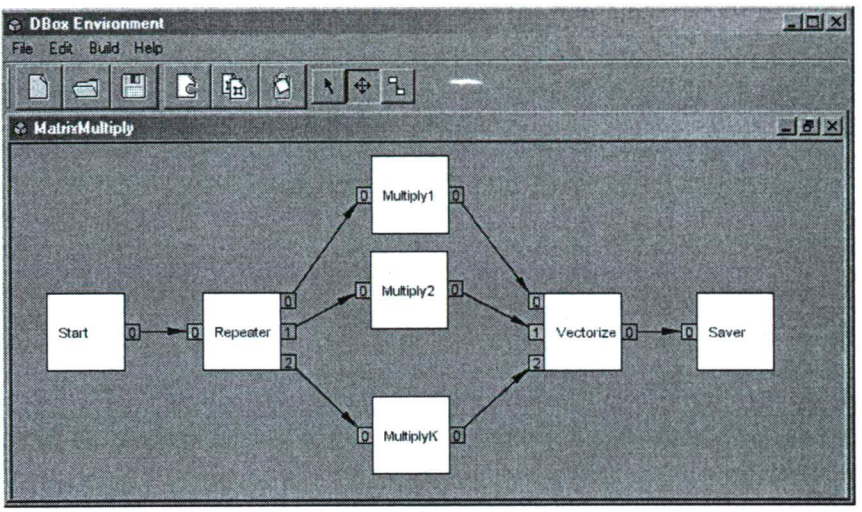


Figure 13: Matrix-multiplier example

10 Related works

- PMLS and GpH are implicit parallel extensions of ML and Haskell respectively [14], on the other hand D-Clean uses explicit coordination structures.

  Opposed to skeleton based languages, D-Clean is designed to implement skeletons of distributed functional computations in the language itself.

- Eden [15, 13] extends Haskell to explicitly define parallel computation. Eden program consits of processes and uses communication channels, and the programmer has explicit control over communication topology. The execution is based on GHC implementation of concurrency, the run-time system controls sending and receiving messages, process placements and data distribution. On the other hand the middleware supporting the implementation of DClean and DBox languages is not language specific, components developed using other languages can be integrated into easily distributed applications.

- Nimo [3] is a visual functional dataflow language, supporting process networks. Nimo allows totally graphic programming only, while DClean and DBox programs can be expressed in textual code form too. Nodes in Nimo are restricted for a fixed set of primitive operations of Haskell prelude, while in DClean nodes Clean expressions are allowed to achieve full power of functional programming at node level. Nimo does not support distributed computing, only concurrent execution is supported.

- JoCaml is an extension of Objective Caml with primitives for network-transparent distributed and mobile programming [7] based on the join-calculus model instead of a pure data flow approach.

Advanced discussion and survey of the dataflow languages can be found in [18]. Data oriented skeletons (like the farm skeleton) can be implemented using primitives which are quite similar to the primitives of dataflow languages.

11 Conclusion and future works

The distributed functional skeletal programming requires higher order coordination structures in order to coordinate the computation of several functional clients in an abstract way. We extended Clean with powerful coordination language elements as tools for description of distributed computation patterns. We proposed a higher level and an intermediate level coordination language, the D-Clean and the D-Box languages. The implementation is based on a multi-paradigm environment, using an object-oriented middleware, which supports the interconnection of client and server programs written in different programming languages.

The high-level coordination language D-Clean is appropriate for definition of functional skeletons at a very high abstraction level. These skeletons can be parameterized by types and by functions. As future work we plan to extend the language with the possibility of parameterization of schemes by distribution strategy and to enable the description of dynamic configurations. We also plan a description of a more formal type system and type constraints. Dynamic creation of the computational nodes, communication channels and dynamic start of the functional components are highly needed at the development of the applications using recursive expressions.

12 Acknowledgements

The authors would like to thank the anonymous referees for their insightful comments, which has helped immensely in improving the quality of the paper.

References

[1] Berthold, J., Klusik, U., Loogen, R., Priebe, S., Weskamp, N.: High-level Process Control in Eden, In: Kosch, H., Böszörményi L., Hellwagner, H. (Eds.): *Parallel Processing, 9th International Euro-Par Conference, Euro-Par 2003*, Proceedings, Klagenfurt, Austria, August 26-29, 2003, Springer Verlag, LNCS Vol. 2790, pp. 732-741.

[2] Best, E., Hopkins, R. P.: B(PN)^2 - a Basic Petri Net Programming Notation, In: Bode, A., Reeve, M., Wolf, G. (Eds.): *Parallel Architectures and Languages Europe, 5th International PARLE Conference, PARLE'93*, Proceedings, Munich, Germany, June 14-17, 1993, Springer Verlag, LNCS Vol. 694, pp. 379-390.

[3] Clerici, S., Zoltan, C.: A Graphic Functional-Dataflow Language, In: Loidl, H.W. (Ed.): *Proceedings of the Fifth Symposium on Trends in Functional Programming*, Ludwig-Maximilians University, 25-26 November, Munich, Germany, 2004, pp. 345-359.

[4] Cole, M.: Algorithmic Skeletons, In: Hammond, K., Michaelson, G. (Eds.): *Research Directions in Parallel Functional Programming*, pp. 289-303, Springer-Verlag, 1999.

[5] Danelutto, M., Di Cosmo, R., Leroy, X., Pelagatti, S.: Parallel Functional Programming with Skeletons: the OCAMLP3L experiment, In: *Proceedings of the ACM, Sigplan Workshop on ML*, Baltimore, USA, September 1998, pp. 31-39.

[6] Dezső B.: *DBox Developer Environment*, Project work documentation, Department of Programming Languages and Compilers, University Eötvös L., Budapest, Hungary, 2005.

[7] Fournet, C., Le Fessant, F., Maranget, L., Schmitt, A.: JoCaml: A Language for Concurrent Distributed and Mobile Programming, In: Johan Jeuring, Simon Peyton Jones (Eds): *Advanced Functional Programming, 4th International School, AFP 2002, Oxford*, Revised Lectures, Springer, LNCS 2638, pp. 129-158, 2003.

[8] Hammond, K., Portillo, A. J. R.: Haskel: Algorithmic Skeletons in Haskell, In: Koopman, P.; Clack, C. (Eds.): *Implementation of Functional Languages, 11th International Workshop IFL'99*, Lochem, The Netherlands, September 7-10, 1999, Selected Papers, Springer Verlag, LNCS Vol. 1868, 2000, pp. 181-198.

[9] Hernyák Z., Horváth Z., Zsók V.: Clean-CORBA Interface Supporting Skeletons, To appear in: *Proceedings of 6th International Conference on Applied Informatics*, Eger, Hungary, January 27-31, 2004.

[10] Hernyák Z., Horváth Z., Zsók V.: Design of Language Elements for Dynamic Distributed Computation of Clean Expressions on Clusters, In: Loidl, H.W. (Eds.) *Proceedings of the Fifth Symposium on Trends in Functional Programming*, Ludwig-Maximilians University, 25-26 November, Munich, Germany, 2004, pp. 257-270.

[11] Horváth Z., Zsók V., Serrarens, P., Plasmeijer, R.: Parallel Elementwise Processable Functions in Concurrent Clean, In: *Mathematical and Computer Modelling* 38, pp. 865-875, Pergamon, 2003.

[12] Kesseler, M.H.G.: *The Implementation of Functional Languages on Parallel Machines with Distributed Memory*, PhD Thesis, Catholic University of Nijmegen, 1996.

[13] Loidl, H.W., Klusik, U., Hammond, K., Loogen, R., Trinder, P.W.: GpH and Eden: Comparing Two Parallel Functional Languages on a Beowulf Cluster In: Gilmore, S. (Ed.): *Trends in Functional Programming*, Vol. 2, pp. 39-52, Intellect, 2001.

[14] Loidl, H-W., Rubio, F., Scaife, N., Hammond, K., Horiguchi, S., Klusik, U., Loogen, R., Michaelson, G.J., Peña, R., Priebe, S., Rebón Portillo, Á.J., Trinder, P.W.: Comparing Parallel Functional Languages: Programming and Performance, In: *Higher-Order and Symbolic Computation* 16 (3), pp. 203-251, Kluwer Academic Publisher, September 2003.

[15] Peña, R, Rubio, F., Segura, C.: Deriving Non-Hierarchical Process Topologies, In: Hammond, K., Curtis, S.: *Trends in Functional Programming*, Vol 3. pp. 51-62, Intellect 2002.

[16] Plasmeijer, R., van Eekelen, M.: *Concurrent Clean Version 2.0 Language Report*, 2001, http://www.cs.kun.nl/~clean/Manuals/manuals.html.

[17] Serrarens, P.R.: *Communication Issues in Distributed Functional Computing*, Ph.D. Thesis, Catholic University of Nijmegen, January 2001.

[18] Wesley M. Johnston, J. R. Paul Hanna, Richard J. Millar: *Advances in dataflow programming languages*, ACM Comput. Surv., ACM Press, New York, USA, 36(1), 2004, pp. 1-34.

[19] Zólyomi I., Porkoláb Z., Kozsik T.: An extension to the subtype relationship in C++ implemented with template metaprogramming, *GPCE '03: Proceedings of the second international conference on Generative programming and component engineering*, Erfurt, Germany, Springer-Verlag New York, 2003, pp. 209-227.

[20] Zsók V., Horváth Z., Varga Z.: Functional Programs on Clusters, In: Striegnitz, Jörg; Davis, Kei (Eds.): *Proceedings of the Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'03)*, Interner Bericht FZJ-ZAM-IB-2003-09, July 2003, pp. 93-100.

13 Appendix

The pseudo code of `DStart expr`:

```
Start w
   # (result1, result2, ..., w) = expr w
   # (channel1, w) = Channel_FIND output_channel_id_1 w
   ...
   # channels = [channel1, channel2,...]
   # w        = split1 channels result_1 result_2 ... w
   = w
```

The pseudo code of `DStop expr`:

```
Start w
   # (channel1, w) = Channel_FIND input_channel_id_1 w
   ...
   # channels = [channel1, channel2,...]
   # (data, w) = join1 channels w
   = expr data w
```

The pseudo code of `split1` and `splitk`:

```
split1 ::[ChannelID]  [a] [b] [c] [d] ... *World -> *World
split1 channels data0 data1 data2 ... w
     # w = SendStream channels!!0  data0 w
     # w = SendStream channels!!1  data1 w
     ...
     = w
splitk :: [ChannelID] [[a]] *World -> *World
splitk [] [] w = w
splitk [ch:channels] [d:data] w
   = SendStream ch d w2
     where
       w2 = splitk channels data w
```

The pseudo code of `join1`:

```
join1 ::[ChannelID] *World -> ([a],[b],[c],...,*World)
join1 channels w
      # (a,w) = ReceiveStream channels!!0 w
      # (b,w) = ReceiveStream channels!!1 w
      ...
      = (a,b,c,...,w)
```

The pseudo code of `joink`:

```
joink:: [ChannelID] *World -> ([[a]], *World)
joink [] w = ([],w)
joink [ch:cs] w  = ( [data : remaining], w4 )
  where
    (data, w3) = ReceiveStream ch w
    (remaining, w4) = joink cs w3
```

Acta Cybernetica 17 (2005) 273–287.

An Approach based on Genetic Algorithms for Clustering Classes in Components

Dan Laurenţiu Jişa*

Abstract

The goal of this work is to create a model that allows identification of the software components (or subsystems according to the unified process terminology) based on the design models, or more exactly, based on the classes diagrams (for the static aspects) and on the interaction diagrams (for the dynamic aspects). The work also presents a genetic algorithm used for the clustering of classes into modules.

Keywords: object-oriented metrics, UML, unified process, genetic algorithms, clustering algorithms.

1 Introduction

The components based development (CBD) process tends to revolutionize the process of applications development, the usage of components opens the possibility of assembling applications from predefined building blocks. A component has to assure a better management of the complex applications, to lead to a decrease of the development costs and to an increase of the flexibility. These goals could be reached through a proper quality level of software components. One of the most important quality indicators for a software module is the reusability. The term "reuse" refers to the utilization of a software product, earlier developed, in a new project or software application. This reutilization can take many aspects:

- any physical component or program code;
- any product, result of a software development process: tools, documentation, models (requirements, analysis or design) etc.;
- any knowledge gains in the earlier projects.

The reutilisation, in a development process, of a software component that already exists, instead of the development of a new one, represents an activity that leads to an increase of product quality and, also, to a decrease of the development costs. These benefits are the consequence of two reasons:

*PhD student, Department of Business Informatics, Academy of Economic Studies, Calea Dorobanţilor 15-17, Sector 1, 71131-Bucharest, Romania, e-mail:dan.jisa@estwest.ro

- the development of new components (modules) is expensive;
- the reused components (modules) are considered as being very well tested, and their maintenance is not very expensive.

From the point of view of the software metrics, the reuse of software products provides many interesting attributes to estimate. For example, these attributes can be: the quantity of reused code in a software product, the reutilisation cost, the reusability of a certain product, module or class etc.

The measurement of the degree to which a software product can be reused, usually takes the name of reusability. Generally, the reusability is defined as the extent to which a software product can be used in other applications.

Often, the goal of reusability measurements is to identify components in large applications. In addition, the goal is often to automatically extract the components. For the domain experts it will be still necessary to assess the modules in order to identify the possible modifications. However, some results show that through the automation of the identification process, the quantity of code which has to be examined by the domain experts will be reduced.

Another important objective is to design and to write a reusable code. When the software development process is mature enough and the software metrics become part of the process carried on in an organisation, then the design metrics become more important for two reasons:

- the measurement process can be automated, so that to have a feedback for the designers;
- the measurement of the reusability, performed over the design model of a software product, allows the early identification of products with a low potential of reusability, when the modification and the refinement are still possible (and the cost is not too high).

The identification of reusable elements is well known as one of the most difficult tasks in software reuse. Although the traditional elements (cod segments, objects) are the most frequently reused, great benefits are obtained when big elements, as the business components, are reused.

According to the components based methodologies, the clustering process of classes into modules must begin as early as possible in the development lifecycle (in the analysis stage). So that, in the unified process ([BRJ99a]) the analysis model is decomposed into analysis packages in two ways: in a top-down manner, as well as a bottom-up manner. In the bottom-up approach, when the model becomes too large, it must be decomposed into many packages, in such a way that, after decomposition, it should have a high cohesion degree among the classes inside the packages and low values for coupling among packages (that is as few relationships as possible among classes that belong to different packages). The analysis packages will evolve in subsystems (this is the concept used by the unified process in order to specify a software component in the design stage), and the subsystems, in their turn, will evolve in physical components (used in components diagrams). Therefore, it can

be asserted that, among the analysis packages, the subsystems of the design stage and the physical components (specified in UML through the component notation), the traceability is provided.

2 Mathematical model

It will be considered a design model consisting of n classes, where $n > 1$ (n is higher than one). It will be desired to cluster the classes into a certain number of components, k (where $k < n$), so there will be obtained values as high as possible for several quality indicators (indicators used to asses the clustering of classes). Till the present moment (for this stage of the work) it was used only one quality indicator, the reusability.

In the book "Designing Object Oriented C++ Applications Using the Booch Method" ([MAR95]), Robert Martin presents several metrics that can be applied for the *class categories*. The equivalent concept in the Universal Modelling Language is the *package*. Therefore, the R. Martin's metrics can be applied in a context of models developed with the unified process and UML, more exactly they can be applied for the subsystems, which represent the concept used to specify the components at a design level (according to the unified process terminology). A basic premise is the desire to build class structures such that a single change to a class will not propagate up and down the class hierarchy without restraint. The class categories are designed to have the property of *closure*. One way of creating subsystems that exhibit closure, is to ensure that the subsystems with the most dependencies on other subsystems also have the greatest resistance to change; those subsystems that are the most changeable, should also have the fewest relationships with other subsystems.

R. Martin's metrics, used in the presented model (and in the algorithm which will be presented in the next section), are the following:

- Relational cohesion – through this metric it is tried to assess the cohesion degree among the classes inside a package. Relational cohesion is defined as the number of relationships between the classes that belong to the package, divided by the number of classes belonging to the same package;
- Afferent coupling – represents the number of classes that depend on the classes belonging to a specified package; it is recommendable to have low values for this metric;
- Efferent coupling – represents the number of classes outside a specified package which depend on the classes belonging to the package; there is recommendable to have low values for this metric;
- Instability – is defined as the ratio between the efferent coupling and the total coupling (defined as the sum between the efferent and afferent coupling) computed for a package;

The elements of the mathematical model will be represented as follows:

- A matrix $M = (m_{ij})$, that represents the inheritance relationships among the classes of the model, where $i = \overline{1,n}$, $j = \overline{1,n}$, and

$$m_{ij} = \begin{cases} 1, & \text{if class } i \text{ is inherited by the class } j \\ 2, & \text{if class } i \text{ inherits class } j \\ 0, & \text{otherwise} \end{cases}$$

- A matrix $A = (a_{ij})$, that represents the association relationships among the classes of the model, where $i = \overline{1,n}$, $j = \overline{1,n}$ (n is the number of classes of the model), and a_{ij}represents the number of association relationships between classes i and j.

- A matrix $D = (d_{ij})$, that represents the dependency relationships among the classes of the model, where $i = \overline{1,n}$ (i takes values between one and the maximum number of classes - n), $j = \overline{1,n}$, and d_{ij}represents the number of dependency relationships from class i to class j.

- A matrix $Msg = (msg_{ij})$, that represents the messages exchanged among the objects (that is the instances) of the model classes, where $i = \overline{1,n}$, $j = \overline{1,n}$, and msg_{ij}represents the number of messages sent by the objects of class i to the objects of class j.

- A matrix $Met = (met_{ij})$, where $i = \overline{1,n}$ and $j = \overline{1,m}$, where m represents the number of internal metric considered.

$$met_{ij} = \text{the value of metric } j \text{ computed for class } i$$

The goal is to find k components (modules), where $1 \leq k \leq n-1$. For each component it will be possible to impose constraints as regarding the number of classes included (for example, $2 \leq dim \leq n/2$).

A solution is represented by a vector $X = (x_i)$, with $i = \overline{1,n}$, where:

$$x_i = \begin{cases} \text{index of the component to which the class } i \text{ belongs} \\ 0, \quad \text{if the class does not belong to any component} \end{cases}$$

Further on there will be presented the formulas used for computing the R. Martin's metrics (relational cohesion, afferent and efferent coupling), based on the model's elements described above.

In order to compute the relational cohesion, the generalization/specialization, association and dependency relationships were considered, as well as the messages exchanged by classes' objects.

The coupling, also, was computed based on the relationships among classes: generalization/specialization, association and dependency. The interaction diagrams, in UML, are used in order to specify the dynamic aspects of the collaborations among classes: the objects of the classes involved in collaboration exchange messages along the links between them. But, according to UML, the concept of link

represents an instance of an association; therefore, the exchange of messages will be done between objects of the classes related by association relationships. In the formulas used to compute the cohesion and the coupling, the messages exchanged by objects are considered in order to have a measurement of the coupling or cohesion intensity.

The relational cohesion, for a component k and a solution X, was computed based on the model elements presented above, as follows:

$$RCoeh(k, X) = \frac{\sum_{i=1}^{n} \sum_{j=1}^{n} rc(k, i, j, X)}{\sum_{i=1}^{n} s(k, i, X)}, \tag{1}$$

where

$$rc(k, i, j, X) = \begin{cases} w_m \times m_{ij} + w_a \times a_{ij} + w_d \times d_{ij} + w_{msg} \times msg_{ij}, \\ \qquad \text{if } x_i = k \wedge x_j = k, \\ 0, \qquad \text{otherwise,} \end{cases} \tag{2}$$

and

$$s(k, i, X) = \begin{cases} 1, & \text{if } x_i = k \\ 0, & \text{otherwise} \end{cases}$$

w_m = the weight assigned to the inheritance relationships;
w_a = the weight assigned to the association relationships;
w_d = the weight assigned to the dependency relationships;
w_{msg} = the weight assigned to the messages exchanged by classes' objects;

As it can be observed in the above formulas, for each relationship type implied in the metrics calculation there was allowed the assignment of weights. This is so because different types of relations among classes have a different meaning as regards the relations' strength (an inheritance relationship represents a stronger relation than a dependency). As a consequence, the software developers could want to assign weight to different types of relations, in accordance with the importance assigned in the clustering process.

The formulas used in order to compute the afferent and efferent coupling for a component k and a solution X, are the following:

$$AC(k, X) = \sum_{i=1}^{n} \sum_{j=1}^{n} ac(k, i, j, X), \tag{3}$$

where

$$ac(k, i, j, X) = \begin{cases} w_m \times m_{ij} + w_a \times a_{ij} + w_d \times d_{ij} + w_{msg} \times msg_{ij}, \\ \qquad \text{if } x_i \neq k \wedge x_j = k \\ 0, \qquad \text{otherwise} \end{cases}$$

$$EC(k,X) = \sum_{i=1}^{n}\sum_{j=1}^{n} ec(k,i,j,X), \tag{4}$$

where

$$ec(k,i,j,X) = \begin{cases} w_m \times m_{ij} + w_a \times a_{ij} + w_d \times d_{ij} + w_{msg} \times msg_{ij}, & \text{if } x_i = k \wedge x_j \neq k \\ 0, & \text{otherwise} \end{cases}$$

The mathematical model, based on the previous elements, is:

$$maximize\, F(X) = \sum_{k} \left[\frac{RCoeh(k,X)}{(AC(k,X) + EC(k,X) + 1) + \frac{EC(k,X)}{AC(k,X) + EC(k,X)}} \right], \tag{5}$$

where

X = represents the searched solution $X = (x_i)$, $i = \overline{1,n}$
It will be necessary to find out X so that $F(X)$ will be maximur
$RCoeh(k,X)$ = represents the relational cohesion for component k (formula 1);
$AC(k,X)$ = represents the afferent coupling for component k (formula 3);
$EC(k,X)$ = represents the efferent coupling for component k (formula 4);

The constraints of the model are:

$$\begin{cases} G(k,X) \geq \dfrac{number\_of\_classes\_in\_module - 1}{number\_of\_classes\_in\_module} \\ AC(k,X) + EC(k,X) > 0 \\ k\min \leq \sum\limits_{i} s(k,i,X) \leq k\max \end{cases}$$

where

$$G(k,X) = \sum_{i}\sum_{j} g(k,i,j,X),$$

and

$$g(k,i,j,X) = \begin{cases} 1, & \text{if } x_i = k \wedge x_j = k \wedge rc(k,i,j,X) > 0 \\ 0, & \text{otherwise} \end{cases}$$

The meaning of the constraints is:

- a subsystem must contain more than one class, and among them there have to be relationships (otherwise the value of the relational cohesion will be zero); it must not contain classes that are not related to the other classes within the subsystem;

- a subsystem must have relationships with the other subsystems (it cannot exist in isolation); this means that the total coupling (efferent or afferent) will be higher than zero;
- for a subsystem, the developer can impose conditions as regarding the number of classes (a minimum and a maximum number of classes).

3 The genetic approach

For the clustering of classes in subsystems (software components) the work proposes the utilization of a genetic algorithm. A genetic algorithm applies ideas taken from the natural selection theory, in order to navigate through a large solutions space.

Successfully applying a genetic algorithm in order to solve a problem implies that:

a) a suitable representation of the solution must be found; a solution will be represented by a chromosome;
b) a fitness function must be established in order to evaluate each solution;
c) the genetic operators, which will be applied in the algorithm, must be established;
d) the values of the algorithm parameters must be established, that is: dimension of the population, the probabilities with which the operators will be applied etc.;

a) The ***representation of a solution*** must be:

- complete;
- valid.

A complete representation assumes that there is a possibility to encode all the solutions for the studied problem. A valid representation assumes that all the codifications are in the solution space. Invalid representations can be used, but the algorithm has to be adapted in order to avoid invalid solutions.

In this paper, for the discussed problem, a solution was encoded as a vector (the vector elements are positive integers), as follows:

- the index of each element represents a class of the model;
- the value of each element is a positive integer and represents the index of the component to which the class belongs.

Each element of a chromosome can take a value between one and the maximum number of components. Through this representation the completeness is assured. Therefore, it will be possible to represent any clustering of classes in modules, through a chromosome. The validity of a chromosome has to be checked out, in case there are imposed constraints regarding the number of classes within a module.

b) The ***fitness function*** quantifies each solution represented by a chromosome and is used as basis for chromosomes selection for mating. For the discussed problem the fitness function is represented by formula (5).

The objective function is built so that it will lead to a clustering in subsystems (components) with low values for coupling (afferent and efferent) and for instability, and high values for relational cohesion.

The function computed for each subsystem (component), is the ratio between the relational cohesion of the classes within the module, and the sum between the coupling (afferent and efferent) and the module instability.

The relational cohesion (formula 1) is computed as the ratio between the number of relationships among the classes within a module and the number of classes belonging to the module, so that it will not be possible to have high values for cohesion (and for the fitness function) if a subsystem contains a high number of classes, but with few relationships among them.

c) The ***operators*** used by the algorithm are the following:

Initialization operator – the initial population is randomly generated. The algorithm was tested using several initialization operators that generate individuals composed by groups of two, three, four and five classes. For each element of a chromosome is generated a positive integer between one and the maximum number of subsystems (in which the model can be divided).

The *selection operator* used by the algorithm is *roulette wheel.*

As regarding the *crossover* operator, the algorithm was tested using several crossover operators. The classical operators used, are the following:

- one-point-crossover;
- two-point-crossover;
- uniform-crossover;

In addition to the above presented operators, it was tested a new one (named ClusterCrossover) for which there have been obtained the good values for the fitness function. The results are presented in section 4.

Further on the new crossover operator will be presented.

The operator works as follows: having two chromosomes selected for crossover, $X_p = (x_{p1}, ..., x_{pn})$ and $X_q = (x_{q1}, ..., x_{qn})$. One point, in the first chromosome, is randomly selected: s_{pi}. The new chromosomes are created as follows:

- the first child is composed of all the positions from Xp equals with x_{pi} and the others positions from Xq;
- the second child takes the remaining values from the two parent chromosomes.

The mutation operators used by the algorithm were *swap-mutator* and a mutation operator presented in the paper "FGKA: A Fast Genetic K-means Clustering Algorithm" ([LLF04]), adapted for the discussed problem, which will be presented further on.

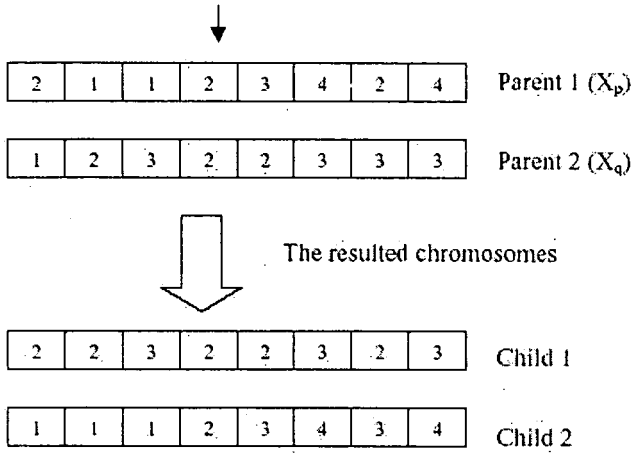


Figure 1: The crossover operation

If it is considered a chromosome $X_p = (x_{p1}, ..., x_{pn})$, the element x_{pi} is replaced by the value $x_{pi'}$, for $i = \overline{1,n}$, where $x_{pi'}$ is a subsystem, randomly selected from 1 to k, with the probability:

$$pmut_t = \frac{1,5 * Dist_{pmax} - Disp_p(i, x_{pi}) + 0.5}{\sum_{j=1}^{k}(1,5 * Dist_{pmax} - Dist_p(i,j) + 0.5)},$$

where t between 1 and k (k is the maximum number of components).

$Dist_{pmax}$ represents the maximum square of the Euclidian distance between class i and one of the subsystems, and $Dist_p(i)$ represents the square of the Euclidian distance between class i and the subsystem x_{pi}.

The square of the Euclidian distance is computed as the square of the difference between the centroid of the subsystem x_{pi}($Centr(x_{pi}, X_p)$)and the weight of the class i ($W(i, X_p)$):

$$Dist_p(i) = [Centr(x_{pi}, X_p - W(i, X_p))]^2$$

The weight W of the class i is computed with the following formula (the ratio between all the relationships between class i and the other classes belonging to the same subsystem, divided to the number of classes within the subsystem, and the relationships between class i and the classes outside the subsystem):

$$W(i, X_p) = \frac{\sum_{j,i \neq j}[cupl\_in(i, j, X_p)/s(i, X_p)]}{\sum_{j,i \neq j} cupl\_out(i, j, X_p)}, \tag{6}$$

where

$$cupl\_in(i, j, X_p) = \begin{cases} w_m \times m_{ij} + w_a \times a_{ij} + w_d \times d_{ij} + w_{msg} \times msg_{ij}, & \text{if } x_{pi} = x_{pj} \\ 0, \quad \text{if } x_{pi} \neq x_{pj} \end{cases}$$

$$cupl\_out(i,j,X_p) = \begin{cases} w_m \times m_{ij} + w_a \times a_{ij} + w_d \times d_{ij} + w_{msg} \times msg_{ij}, \\ \quad \text{if } x_{pi} \neq x_{pj} \\ 0, \quad \text{if } x_{pi} = x_{pj} \end{cases}$$

The formula for the centroid of the subsystem x_{pi} is:

$$Centr(i,X_p) = \frac{\sum_i class\_weight(i,t,X_p)}{\text{number of classes within module } t}, \tag{7}$$

where

$$class\_weight(i,t,X_p) = \begin{cases} W(i,X_p), & t = x_{pi} \\ 0, & t \neq x_{pi} \end{cases}$$

d) The ***algorithm parameters*** were found as follows:

- the initial dimension of the population: 20 x n (where n is the number of classes within the model);
- the crossover probability: 0.9;
- the mutation probability: 0.01;
- the overlapping degree of the populations: 20%;
- number of generations: 50 x n;

The algorithm involves the following steps:

1. The initial population is generated. There are created fixed length strings (the length of a string is equal to the number of classes within the model). The strings are randomly initialized using one of the initialisation operators described above.
2. There are selected the individuals for mating using roulette wheel operator.
3. One of the crossover operators described above is applied.
4. One of the mutation operators presented above is applied.
5. The old population is replaced by the new one: the worst individuals are removed from the temporary population, in order to return the population to its original size.
6. If the required number of iterations is not reached, then the algorithm will continue with step 2.

4 Case study and results

The results that will be presented further on, were obtained as a result of running the algorithm on a model composed of 17 classes. This model, showed in figure 2, is extracted from a larger project, in which a B2B application is developed: an intermediary site that takes the requests (the orders) from dealers and sends them to manufacturers.

At present, the information about the model is extracted manually and introduced into a database in order to be used by the algorithm. Further on, an application will be developed that will be able to extract the information about the UML model from an XMI (XML Metadata Interchange) file.

The following image presents the cluster of classes proposed by the algorithm (in the best case).

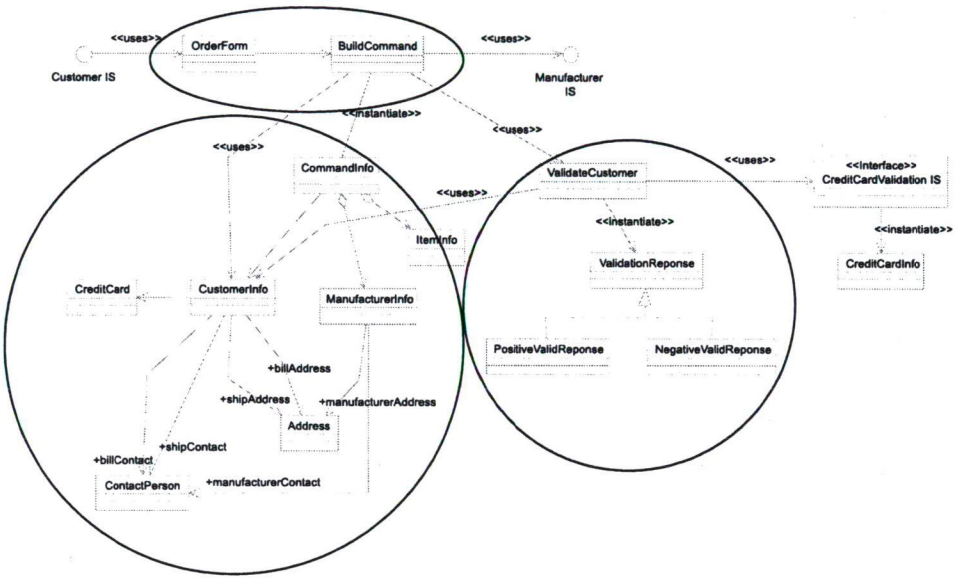


Figure 2: The clusters of classes proposed by the algorithm

The following figure represents the results obtained as a consequence of applying the swap-mutator operator together with the uniform, two-points and cluster-crossover (the new operator tested), for initial population which contains individuals formed by clusters of three classes.

As it can be observed from the above picture, the best results were obtained for the ClusterCrossover operator.

Figure 4 presents the results obtained as a consequence of applying the same crossover and mutation operators as in the previous figure, but with an initial population which contains individuals formed by clusters of four classes (the initialization operator is different).

Again, like in the first case, the best fitness function value is obtained in a smaller number of iterations for the ClusterCrossover operator, than for uniform and two-point crossover.

In the following picture the same operators are applied as in the previous case, but with the initial population which contains individuals formed by cluster of five classes.

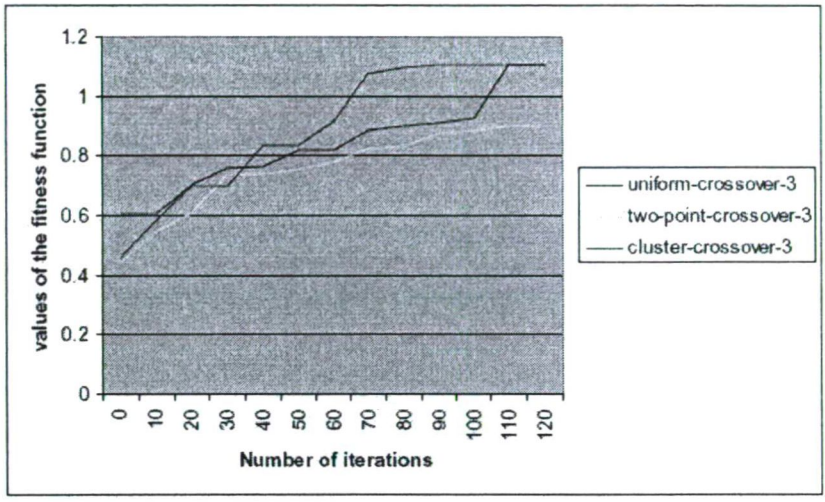


Figure 3: Results for an initial population composed by individuals formed by clusters of three classes

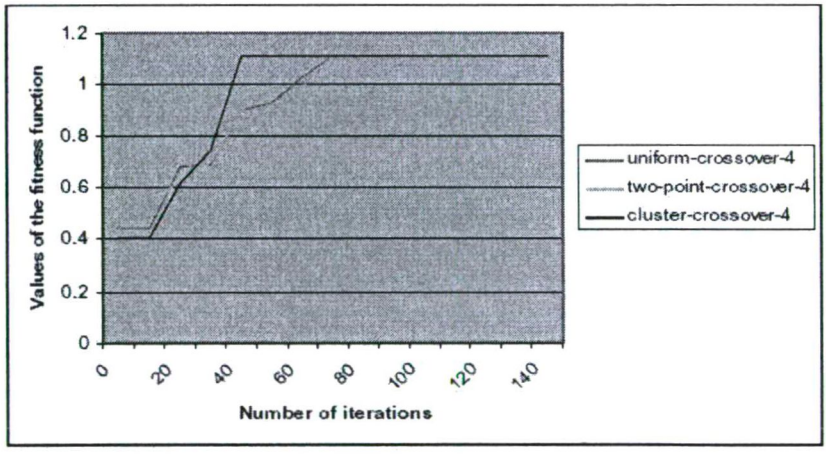


Figure 4: Results for an initial population composed by individuals formed by clusters of four classes

In figure 6 there are presented the results obtained as a consequence of applying the FGKA-mutator operator together with the uniform, two-points and cluster-crossover (the new operator tested), for the initial population which contains individuals formed by clusters of two classes. It can be asserted that if the algorithm starts with a large number of subsystems that must be reduced during the evolution of the algorithm, then the FGKA-mutator is more suitable than the swap-mutator operator.

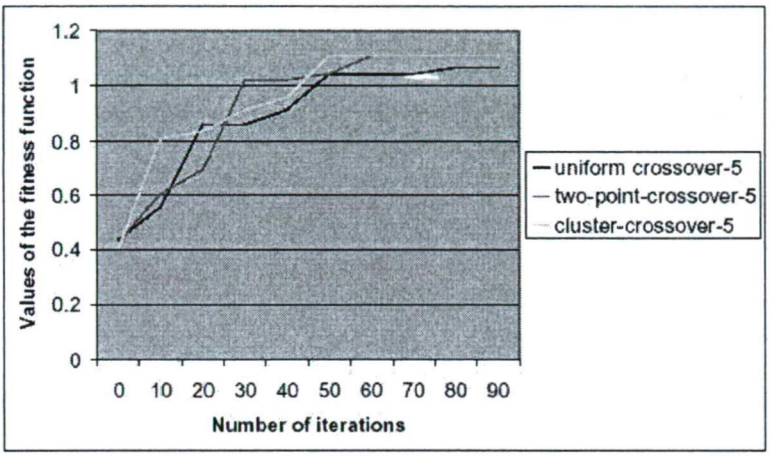


Figure 5: Results for an initial population composed by individuals formed by clusters of five classes

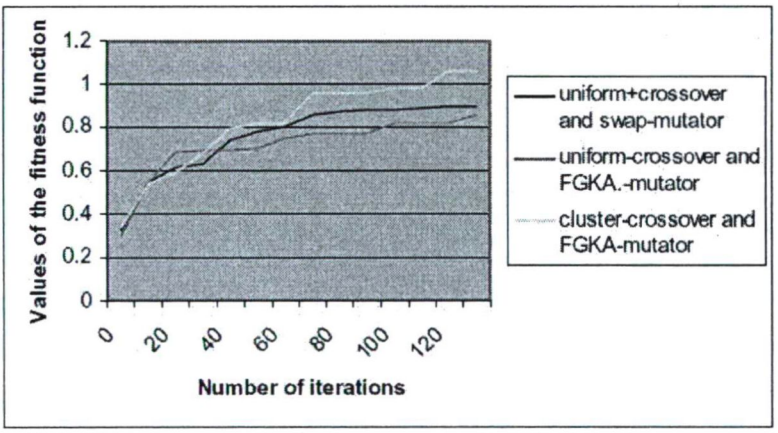


Figure 6: Results of applying FGKA-mutator together with the uniform, two-points and cluster-crossover

5 Conclusions and future work.

The structure of the software systems can be very complex. A design model for a medium size application can include tens of classes (even more than one hundred), the objects (the instances) of a class could be included in more than one interaction diagram. Therefore, it is difficult enough for a developer to find the optimum distribution of classes among the modules that compose the application, so that the reusability should be as easy as possible and should involve a minimum number of modifications (the ideal case is to have not any modification) within the module.

The goal of the algorithm proposed in this work is to help software developers in order to reach a decision as close as possible to the best one (to find the optimal distribution of classes among subsystems).

There are, already, in the speciality literature, approaches that try to identify software components (to group classes in components), but they are applicable during the implementation phase (on the source code). Therefore, in [ETZ97] is presented an algorithm for the automated identification of reusable software components, which collects the metrics from a C/C++ source code. The algorithm is based on metrics like LCOM, average number of comment lines per method etc., that it is applicable only for the implementation phase.

Another approach, presented in [JCI01], achieves a clustering of classes (using a clustering algorithm) based on coupling intensity between them. In a second phase there are applied heuristics in order to find the best values for several quality indicators. In contrast with this approach, the algorithm presented in this work achieves the clustering of classes into subsystems, having as goal a high level of reusability for them. The clustering is based on several architectural metrics (R. Martin's metrics), that can be applied against the subsystems (packages). In addition to the well known metrics, like cohesion and coupling, there is also considered the module stability. According to R. Martin, "the dependencies between components in a design should be in the direction of stability" and "a component should only depend upon components that are more stable than it".

The approach presented in this work is addressed to the design phase because it was considered that the impact of modifications, involved by the cluster of classes proposed by the algorithm, is less than during the implementation phase. The future work will consist of:

- New metrics will be included in the objective function, metrics which can be computed at design model level, like the following: depth of inheritance tree (DIT), number of children (NOC) etc.

- Modalities (indicators) will be defined for the assessment of the impact over the final product, as a result of applying the algorithm in the design phase (how evolved the identified subsystems in physical components, implemented using the new technologies).

- The modification of the mathematical model and of the algorithm, in order to be possible to assign a class to several packages. In practice it is possible to have situations in which, in order to reduce the dependency between modules, it should be necessary to introduce new classes, even if the information managed by these is redundant. Therefore, a solution will be represented as a matrix (n x n, where n is the number of classes) in which a matrix element, n_{ij}, will contain the index of the subsystem to which both classes (i and j) belong, or 0 if the classes don't belong to the same subsystem.

References

[BRJ99a] Booch G., Raumbaugh J., Jacobson I., *The Unified Software Development Process*, Addison/Wesley 1999.

[BRJ99b] Booch G., Raumbaugh J., Jacobson I., *The Unified Modelling Language. User Guide*, Addison/Wesley 1999.

[CHK94] Chidamber S., Kemerer C., *Metrics Suite for Object Oriented Design*, IEEE Transaction on Software Engineering, vol.20, No.6, 1994.

[ETZ97] Etzkorn, L. H., *A Metrics-Based Approach to the Automated Identification of Object-Oriented Reusable Software Components*, http://www.cs.uah.edu/~letzkorn/disserta.pdf, 1997.

[JCI01] Hemant Jain, Naresh Chalimeda, Navin Ivaturi, Balarama Reddy, *Business Component Identification – A Formal Approach*, Fifth IEEE International, Enterprise Distributed Object Computing Conference, 2001.

[LLF04] Lu, Y., Lu S., Fotouhi F., *A Fast Genetic K-means Clustering Algorithm*, http:// www.cs.wayne.edu/~shiyong/papers/sac04.pdf, 2004

[KHA01] Khaled El Emam, *Object Oriented Metrics: A Review of Theory and Practice*, http://citeseer.nj.nec.com/479219.html, 2001.

[MAR02] Marinescu, R., *Principles of Object-Oriented Design*, http://labs.cs.utt.ro/labs/ip2/html/2002/lectures/3/lecture3.pdf

[MAR95] Martin, R., *Designing Object-Oriented C++ Applications Using the Booch Method.*, Prentice Hall, Englewood Cliffs, NJ, 1995.

[MIC96] Michalewicz Z., *Genetic Algorithms + Data Structures = Evolution Programs*, Springer-Verlag, Heidelberg, 1996.

[MRC98] Marchesi, M., *OOA Metrics for the Unified Modelling Language*, Proceedings of the 2end Euromicro Conference on Software Maintenance and Engineering, 1998, pp. 67-73.

[XHC99] Tao Xie, Huang Huang, Xiangkui Chen, *Object Oriented Software Metrics Technology*, http://www.cs.washington.edu/homes/taoxie/ RicohmiddleReport.pdf , 1999.

Acta Cybernetica 17 (2005) 289–310.

An approach for compacting XMI documents

Miklós Kálmán*

Abstract

One of the most common formats for storing information is XML. It is used in many areas, with its spectrum expanding day by day. A big drawback of the XML format is that the documents can be quite large. This causes problems wherever size is an important issue, for example in embedded systems or whenever the document has to be transferred over a network.

Another widely used format is XMI (XML Metadata Interchange), which is derived from the XML format. Since XMI is an extension of XML the same problems are inherited. A Metalanguage called SRML was defined which provided a good solution to describe the relationships between XML attributes making the compacting of XML documents possible. The main idea behind our paper is to extend the SRML definition in such a way that it supports the XMI environment. This results in a method for compacting XMI documents using semantic rules.

Introduction

It has become an accepted fact that in the electronic age information exchange and storage is quite important. One of the most common formats for this is XML[3]. This format is rather versatile making it a good choice for common information exchange. More and more applications are able to store information in this format. The application areas span military use[7], medical science(human genome mapping [5], component modelling [6]. If the growth continues at this rate, XML documents will span every area in computing.

XML documents can be quite large, but in many cases systems can only handle smaller files, like in the case of embedded systems. Size is an important issue also when the files have to be transferred over the internet. One solution to overcome this issue is to use general compressors (e.g: gzip) or XML compressors like XMill[4]. Unfortunately the compressed file may still be too large.

In the XML environment the attributes often have some sort of relationship, therefore they may be calculated from each other. To store these calculation rules a metalanguage called SRML[11] (Semantic Rule Metalanguage) was defined. It enabled the compaction of XML documents.

*Department of Software Engineering, Aradi vértanuk tere 1., H-6720 Szeged, Hungary, +36 62 544143, email: kalman@inf.u-szeged.hu

One of the most widespread applications for XML is the XMI format. The term XMI[13] stands for XML Metadata Interchange. It is an XML application that is used to manage the standardized interchange of object models and metadata among groups working in team development environments using tools and applications from various vendors. XMI can also be used to exchange information about data warehouses. XMI is based on three industry standards: XML[3], UML[14] (Unified Modelling Language), and MOF[16] (an OMG modelling and metadata repository standard). The architecture enables tools to share metadata programmatically using XML or CORBA[15] interfaces specified in the UML or MOF standards. Since XMI is an extension of XML it inherits its problems also. Size becomes and issue in the XMI environment as well.

For this reason we wish to extend the SRML definition to support the storage of calculation rules in the XMI environment. The SRML language is based on describing relationships between attributes. The current SRML format for example cannot describe calculation rules which are based on text elements. In XMI documents it is quite common to have text elements, which can and may be calculated. The article describes how the SRML language can be extended to provide a method for XMI compaction. This extension makes the SRML metalanguage more generic and may be applied to other XML based formats as well. We have implemented an XML compactor in [10], which provides an effective way of compacting XML documents. With the help of this extension to the SRML metalanguage it can be modified to handle XMI compaction as well.

In the following sections first some background knowledge will be provided. Then we will show how the SRML metalanguage is extended using examples to illustrate how it can be used. Afterwards some areas will be described, where this method can be of great use. Finally, we round off the paper by mentioning related works, a brief summary and topics for future study.

1 Preliminaries

In this section a basic introduction to XML files will be given as well as the XMI format. The necessary preliminaries for Attribute Grammars will be presented. This will be needed to better understand parts in the subsequent sections.

1.1 XML

The first concept that must be introduced is the XML format. The complete XML description can be found in [3] and [17]. XML documents are quite similar to *html* files, as they are both text-based. The components in both are called *elements*, which may contain further *elements* and/or text, or they may be left empty. *Elements* may have attributes like the html anchor tag *a* attribute of *href* elements in html files. *Figure 1* describes a simple example, which stores automobiles in an XML format. Each automobile has the following attributes: Make, Model, Year, Color, NetPrice, Tax, SalesPrice.

```
<XML>
 <Auto Make="Ford" Model="Mondeo" Year="2000" Color="Black"
       NetPrice="25000" Tax="15" SalesPrice="28750"/>
 <Auto Make="Opel" Model="Astra" Year="2004" Color="Papyrus"
       NetPrice="20000" Tax="15" SalesPrice="23000"/>
 <Auto Make="Volvo" Model="S40" Year="2000" Color="Red"
       NetPrice="30000" Tax="15" SalesPrice="34500"/>
 <Auto Make="Fiat" Model="Stilo" Year="2000" Color="Red"
       NetPrice="18000" Tax="15" SalesPrice="20700"/>
 <Auto Make="Toyota" Model="Corolla" Year="2000" Color="Red"
       NetPrice="24000" Tax="15" SalesPrice="27600"/>
</XML>
```

Figure 1: Example for automobile storage in XML

1.2 DTD

The term DTD[3] is short for Document Type Definition. The purpose of a DTD is to define the legal building blocks of an XML document. It defines the document structure with a list of legal elements. The DTD can be viewed as he grammar of a class of documents. A DTD can be declared inline in the XML document, or as an external reference. A DTD contains markup declarations. Each declaration can be either an element type declaration, an attribute-list declaration, an entity declaration, or a notation declaration. Below some explanation will be given for the first two declaration types, since this article uses only these.

- **Element type declaration:** The element structure of an XML document may be constrained using element type and attribute-list declarations for validation purposes. An element type declaration constrains the element's content. For example if an XMI/XML file can only contain <expr> elements it can be listed here.

- **Attribute-list declaration:** Attribute-list declarations specify the name, data type, and default value (if any) of each attribute associated with a given element type. The definition of an attribute-list can be done using the <!ATTLIST ..> keyword. For example if the <expr> element can have a "type" attribute it would be declared the following way:
  <!ATTLIST expr type CDATA #REQUIRED> The word #REQUIRED denotes that the attribute must exists on all "expr" elements. If the word #IMPLIED is used it is not obligatory to have the given attribute present in all cases.

If an XML document uses a DTD file then all of its elements must conform to the DTD specification. Using a DTD it is possible to define a standard form for interchanging data. An application can use a standard DTD to verify that data that it receives from the outside world is valid, but it can also verify that the data it is storing conforms to this specification.

1.3 XMI

The XMI format (XML Metadata Interchange) is an extension of MOF[16] into an XML environment. It is an OMG[12] standard that defines the rules for generating an XML DTD from a metamodel. The MOF (Meta Object Facility) was defined before XML became so widespread. However, the technology-neutral nature of the MOF Core made it relatively easy to produce a mapping from the MOF Core's elements to XML so that, given a metamodel, a Document Type Definition (DTD) could be generated. This DTD[3] can be used to stream models that conform to the metamodel.

The MOF Core, as a subset of UML[14], is object oriented; XML is not. Good object models make liberal use of object orientation and subtyping, therefore they should not be restricted in any way. The architects of XMI wished to avoid having DTD elements repeat all the properties of all their ancestors since that would make it quite cumbersome to render typical object models to DTDs

The UML DTD is the most widely used XMI DTD. UML is not only a notation, but the official specification has a complete MOF-compliant metamodel. It is MOF-compliant because its elements are defined via the constructs of the MOF Core. This metamodel was fed into an XMI DTD generator to produce the UML DTD used by tools to export and import UML models.

XMI provides an open interchange between applications written by different vendors. These applications include:

- Design Tools: these tools include object-oriented UML tools like Rational Rose
- Development tools including integrated development environments like IBM VisualAge for Java and Microsoft Visual Studio .NET
- Databases, Data Warehouses and Business Intelligence tools like IBM DB/2, Visual Warehouse, Intelligent Miner for Data and Oracle 8i
- Software assets like program source code (C/C++, Java) and CASE tools such as TakeFive SniFF+
- Repositories, including IBM VisualAge TeamConnection and Unisys Universal Repository
- Reports, report generation tools, documentations tools, and web browsers

In *Figure 2* the XMI Open Interchange can be seen for the applications mentioned above. If a vendor wishes to participate in the architecture they only need to add XMI support to their product to leverage access to all the other tools.

The XMI version of the example described in *Figure 1* can be created. In *Figure 3* the DTD of the XMI example is presented and *Appendix A* describes the automobile example in an XMI form.

The MOF-compliant metamodel used in this example can be seen in *Figure 4*. It is a simple class with the attributes "Make", "Model", "Color", "NetPrice","Tax","SalesPrice".

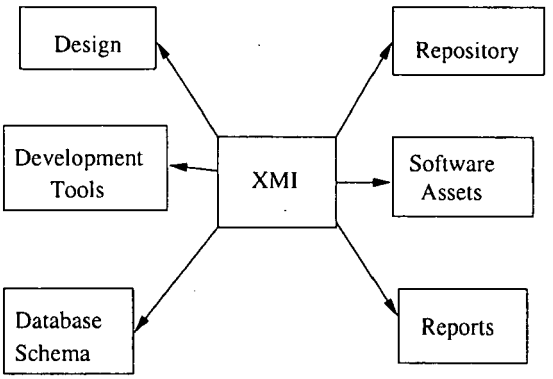


Figure 2: Open interchange with XMI

```
<!ELEMENT Auto (Auto.Make, Auto.Model, Auto.Year, Auto.Color,
                Auto.NetPrice, Auto.Tax, Auto.SalesPrice, XMI.extension*)?>
<!ATTLIST Auto %XMI.element.att; %XMI.link.att;>
<!ELEMENT Auto.Make (#PCDATA | XMI.reference)*>
<!ELEMENT Auto.Model (#PCDATA | XMI.reference)*>
<!ELEMENT Auto.Year (#PCDATA |XMI.reference)*>
<!ELEMENT Auto.Color (#PCDATA | XMI.reference)*>
<!ELEMENT Auto.NetPrice (#PCDATA | XMI.reference)*>
<!ELEMENT Auto.Tax (#PCDATA | XMI.reference)*>
<!ELEMENT Auto.SalesPrice (#PCDATA | XMI.reference)*>
```

Figure 3: The DTD for the XMI automobile example.

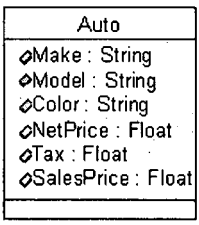


Figure 4: The MOF-compliant metamodel used in the XMI automobile example

1.4 Attribute Grammars

Another key concept that should be mentioned is that of Attribute Grammars. Attribute Grammars are based on the context-free grammars. Context Free (CF) Grammars can be used to specify derivation rules for structured documents. A *CF Grammar* is a four tuple $G = (N, T, S, P)$, where N is the set of nonterminal

symbols, T is a set of terminal symbols, S is a start-symbol and P is a set of syntactic rules. It is required that at the left side of every rule only one nonterminal can be present. Given a grammar, a derivation tree can be generated based on a specific input. The grammar described below specifies the format of a simple numeric.

```
N = { expr, multexpr, addexpr, num }
S = expr      T ={"ADD","MUL",NUM}
P :
 (1) expr      -> num                  (5) addexpr  -> expr "SUB" expr
 (2) expr      -> multexpr             (6) multexpr -> expr "MUL" expr
 (3) expr      -> addexpr              (7) multexpr -> expr "DIV" expr
 (4) addexpr   -> expr "ADD" expr      (8) num      -> NUM
```

An *Attribute Grammar* contains a CF grammar, attributes and semantic rules. The precise definition of Attribute Grammars can be found in [2] [9]. In this section only those definitions will be mentioned which may be needed to understand the later parts of this article.

An attribute grammar is a three tuple $AG = (G, AD, R)$, where

1. $G = (N, T, S, P)$ is the given context-free grammar.

2. $AD = (Attr, Inh, Syn)$ is a description of attributes. Each grammar symbol $X \in N \bigcup T$ has a set of attributes $Attr(X)$, where $Attr(X)$ can be partitioned into two disjoint subsets denoted by $Inh(X)$ and $Syn(X)$. $Inh(X)$ and $Syn(X)$ denote the inherited and synthesized attributes of X, respectively. We will denote the attribute a of the grammar symbol X by $X.a$.

3. R orders a set of evaluation rules (called *semantic rules*) to each production, as follows: Let p: $X_{p,0} \ldots X_{p,n_p}$ be an arbitrary production of P. An attribute occurrence $X_{p,k}.a$ is said to be a *defined occurrence* if $a \in Syn(X_{p,k})$ and k=0, or $a \in Inh(X_{p,k})$ and $k > 0$. For each defining attribute occurrence there is exactly one rule in R(p) that determines how to compute the value of this attribute occurrence. The evaluation rule defining attribute occurrence $X_{p,k}.a$ has the form: $X_{p,k}.a = f(X_{p,k_1}.a_1, \ldots, X_{p,k_m}.a_m)$.

The example in *Figure 5* shows an AG for computing the *type* of the simple numeric expression described above.

In the example the "type" of *addexpr* is *real* if the first or the second *expr* has a *real* type; otherwise it is *int*.

An analogy between AG and XML documents can be discovered. In Attribute Grammars, the Nonterminals correspond to the elements in the XML document. Syntactic Rules are presented as an element type declaration in the DTD of the XML file. An attribute specification in the AG corresponds to an attribute list declaration in the DTD. A key concept in Attribute Grammars which has no XML counterpart are the semantic rules. It might be useful to apply these semantic rules in the XML environment as well. Keeping this concept in view the SRML[11] metalanguage was introduced. It provides XML documents with the support for semantic rules. The metalanguage is based on the analogy described above.

```
(1) expr     -> num expr.type=num.type;
(2) expr     -> multexpr expr.type=multexpr.type;
(3) expr     -> addexpr expr.type=addexpr.type;
(4) addexpr  -> expr "ADD" expr
    addexpr.type=(expr[1].type=="real"|| expr[2].type=="real")? "real":"int";
(5) addexpr  -> expr "SUB" expr
    addexpr.type=(expr[1].type=="real"|| expr[2].type=="real")? "real":"int";
(6) multexpr -> expr "MUL" expr
    multexpr.type=(expr[1].type=="real"|| expr[2].type=="real")? "real":"int";
(7) multexpr -> expr "DIV" expr multexpr.type="real";
```

Figure 5: An example for the semantic rules of a numeric expression.

2 An approach for XMI compaction

In the previous section it was shown that there is a relationship between XML and AG. This analogy also applies to XMI, as it is an extension of XML. The SRML[11] metalanguage provided a method for compacting XML documents. This metalanguage had its weaknesses. One of these was that it was impossible to reference text nodes, whereas most XML documents use this type of representation alot. If we could extend this metalanguage to be more generic it would be possible to store rules which enable XMI compaction. This section will detail how SRML needs to be extended. An example of XMI compaction will also be provided. This extension would enable our XML compactor [10] to handle XMI files as an input, making the XMI compaction possible.

2.1 Extending the SRML metalanguage

An XML-based metalanguage called SRML[11] (Semantic Rule Meta Language) has been defined to describe semantic rules. With the help of this metalanguage XML files could be compacted using semantic rules. The current SRML definition only allows the description of attribute based rules. It has no way of describing rules which contain references to text elements. This was a restriction in the current SRML definition. In an XMI environment using these text elements as a basis for rule creation is a crucial aspect. In order to provide a way for describing such rules the SRML metalanguage has to be extended. This section will detail how this extension can is accomplished.

The following examples will demonstrate why an extension is required. The example in *Figure 6* contains three attributes x,y,z. The z attribute can be calculated by adding x and y together. The SRML for this XML can be seen in *Figure 7*. The keyword *srml:root* refers to the root of the rule, in this case the *Number* element.

```
<XML>
    <Number x="13" y="10" z="23"/>
</XML>
```

Figure 6: A simple XML containing three attributes

```
<SRML>
 <rules-for root="Number">
  <rule element="srml:root" attrib="z">
   <expr>
    <binary-op op="add">
     <expr><attribute element="srml:root" attrib="x"/></expr>
     <expr><attribute element="srml:root" attrib="y"/></expr>
    </binary-op>
   </expr>
  </rule>
 </rules-for>
</SRML>
```

Figure 7: The SRML rules for *Figure 6*

If we introduce a new attribute called "reftype" into the elements rule and attribute it makes it possible to describe references to text nodes. A text node refers to a value being stored between two element tags (e.g. <value>10</value>) instead of storing them as attributes (e.g: <node1 value="10"/>). The value inside the text node is not important, it can be text or numeric. This reference is a vital part of XMI compaction, since most references are text-node based in an XMI environment. In *Figure 8* the XMI form of the XML defined in *Figure 6*.

```
<XMI>
    <Number>
     <Number.x>13</Number.x>
     <Number.y>10</Number.y>
     <Number.z>23</Number.z>
    </Number>
</XMI>
```

Figure 8: A simple XMI containing three attributes

After extending the SRML metalanguage *Figure 9* shows the SRML rules that can now be defined for *Figure 8*.

```
<SRML>
 <rules-for root="Number">
  <rule element="srml:root" attrib="z" reftype="text">
   <expr>
    <binary-op op="add">
     <expr><attribute element="srml:root" attrib="x" reftype="text"/></expr>
     <expr><attribute element="srml:root" attrib="y" reftype="text"/></expr>
    </binary-op>
   </expr>
  </rule>
 </rules-for>
</SRML>
```

Figure 9: The extended SRML rules for *Figure 8*

It is now also possible to combine references between attributes and text nodes. Considering the example in *Figure 10* new types of rules can be defined. Earlier

it was not possible to define such rules. The example contains a *total* attribute, which can be calculated from the three text nodes that the element has.

```
<XMI>
    <Number total="46">
     <Number.x>13</Number.x>
     <Number.y>10</Number.y>
     <Number.z>23</Number.z>
    </Number>
</XMI>
```

Figure 10: A simple XMI with combined attribute types

The modified SRML rule set can be seen below. It contains both text and attribute references.

```
<SRML>
 <rules-for root="Number">
  <rule element="Number" attrib="z" reftype="text">
   <expr>
    <binary-op op="add">
     <expr><attribute element="Number" attrib="x" reftype="text"/></expr>
     <expr><attribute element="Number" attrib="y" reftype="text"/></expr>
    </binary-op>
   </expr>
  </rule>
  <rule element="srml:root" attrib="total" reftype="attrib">
   <expr>
    <binary-op op="add>
     <expr>
      <binary-op op="add">
       <expr><attribute element="Number" attrib="x" reftype="text"/></expr>
       <expr><attirubte element="Number" attrib="y" reftype="text"/></expr>
      </binary-op>
     </expr>
     <expr><attribute element="Number" attrib="z" reftype="text"/></expr>
     </binary-op>
    </expr>
  </rule>
 </rules-for>
</SRML>
```

The complete modified SRML description can be seen in *Appendix G.* Using the "reftype" extension the compactor would know where to look for the given value. It would either look in an attribute or in a node containing the value as text. Another approach would be to convert all value occurrences to a standard form. This form can be the attribute form, since the SRMLTool[10] can handle these easily, however this parsing would take up more time, compared to just telling the compactor to look for the value elsewhere.

2.2 Compacting XMI documents

After the SRML extension has been described in the previous section now it is possible to define rules for XMI text-nodes, thus making XMI compaction possible.

This section will provide an example for this. First an SRML description will be provided for the example shown in *Appendix A*. The set of SRML rules can be found in *Appendix B*. Before showing the result of the compaction first some explanation will be provided for the rules mentioned in the Appendix. It can be noticed that the "SalesPrice" can be calculated from the "NetPrice" by adding the "Tax" as a percent value. The example contains a constant "Tax" value of 15%. Another rule that can be written is that the "Color" of the car is usually "Red" if the "Year" is 2000. If the rule does not match the text element it will not be removed since then it cannot be restored later. When the SRML rules described in *Appendix B* are applied to the example detailed in *Appendix A* the XMI document can be compacted (see *Appendix C*). The original input XMI was 1788 bytes and the compacted XMI became 1256 bytes. This resulting XMI file could be compacted to 70.2% of the original file.

Using a tool called Columbus[8] makes it possible to create XML/XMI files from C++ programs. It is a reverse engineering tool that analyzes the structure of the code and creates an XML/XMI output, which makes it possible to view the relationship between functions, parameter references...etc. If we use the code sniplet described in *Figure 11* Columbus can generate an XMI output from it. This XMI output is partially shown in *Appendix D*.

Examining the output XMI it can be seen that there are some rules that can be described. One of these is that the *Core.DataType* is usually referencing id_dt_1. Another rule that can be written is that if the *ModelElement.name* ends with *Return* then the *Parameter.kind* is *return*, otherwise it is *inout*. It is also visible that the *ModelElement.visibility* is always *public*. Using these rules it is possible to create the SRML file, which can be seen in *Appendix E*. The original input file size is 14439 and the resulting compacted XMI file becomes 11263, achieving a 12% compaction. The output of the compaction can be seen in *Appendix F*.

Compacting XMI files can be quite effective, since they usually contain many values which can be described using semantic rules through SRML. Some of these have been mentioned earlier and can be seen in *Appendix E*. There are however other rule types which can be described using the SRML metalanguage. One of these would be the path name for the object. These path names can be rather long and are usually repetitive, since they are mostly part of the same object base. If for instance the XMI file is referencing objects in "*/home/Development/ProjectX/...*" then it is possible to create concatenation rules for it. The rule would contain the base path name and concatenate the actual attribute value to the end, making the compaction effective.

One might ask why isn't it possible to describe element names using SRML. For example in the CPPML example described before the element prefix "Foundation.Core." is always present. It might be possible to create an extension which could refer to these types of rules, however this would make the compacted XMI file no longer comply to the DTD. So the alternate solution would be to create a DTD modification as well. This however is only theoretical, it requires future research.

```
class Math
{
      public:
            float result;

      public:
            float GetResult()
            {
               return result;
            }

            float Add(float a, float b)
            {
               result = a+b;
               return result;
            }

            float Mul(float a, float b)
            {
               result = a*b;
               return result;
            }

            float Div(float a, float b)
            {
               if (b==0)
                  result = 0;
               else
                   result = a / b;
               return result;
            }

            float Avg(float a, float b)
            {
               result = Div( Add(a,b), 2);
               return result;
            }
...
```

Figure 11: A simple C++ program sniplet

3 Applicability of the method

The method introduced in this article can be applied in many fields of computing. One of these areas is the field of Relational Databases. The reason why XML/XMI compressors are not effective in this field is that the file has to be decompressed completely in order to access parts of the document. Compaction on the other hand is a much more feasible approach, since the SRML file contains the calculation rules and the compacted XMI/XML file is not in a binary format. If a query is placed against an attribute/text-node of the document only the parts that are affected to respond to the query need to be decompacted. This provides a very optimal solution, since the document can be stored in a compacted form, making the resource requirements much smaller. If a node/attribute has to be added to the

file it is added, then when the file is to be closed it would be compacted once again. We are planning to implement a library which would serve as a layer between the user and the compacted XML/XMI document. In *Figure 12* the architecture of the proposed method is shown.

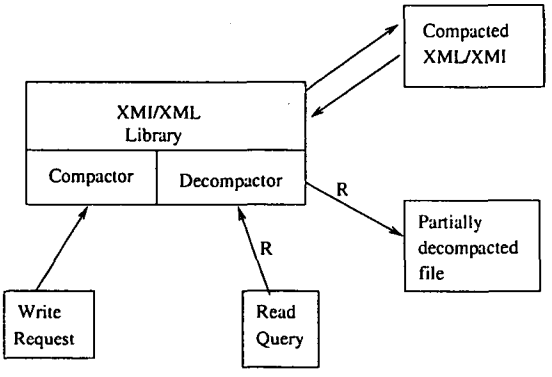


Figure 12: The demand-driven XML/XMI compactor library architecture

4 Related Work

After describing our method it must be mentioned what other research has been done in this area. These articles contain parts which are similar to our approach, but not identical.

The first notion of adding semantics to the XML environment was introduced in [1]. It starts off with a brief introduction to XML. The paper provides a method for transforming the element description of DTD into EBNF formal rule description. Afterwards it introduces its own SRD (Semantics Rule Definition). The reason why we didn't try to extend SRD instead of SRML is that in SRD the attribute definition of elements with a + or * sign is defined in a different way from the ordinary attributes definition and can only reference the attributes of the previous and subsequent element. This would make referencing text elements or regular expressions quite hard to accomplish.

The theory of compacting XML documents using SRML was published in [11]. This definition is quite durable and easily extendible. This is the reason why we chose to extend it into the XMI environment. Once the extension is completed it can be used to compact both XML and XMI documents, which makes it a very valuable asset.

We have implemented an XML Compactor[10] which uses SRML rules to compact XML documents. Our implementation of the XML compactor could achieve a 30% compaction on larger XML files. Using the extension mentioned in this article now it is possible to compact XMI documents with minor modifications to the code.

XMill[4] is an XML compressor. It creates a binary output, therefore we cannot extend it. However if we first compact and XMI file and then compress it with XMill, then the size can be reduced considerably. This means that our method can increase the efficiency of third party compressors.

5 Summary and future work

XML documents have become very widespread. They span many areas of computing. The problem is that they can be quite large at times. The SRML metalanguage is able to store rules which describe how attributes can be calculated from each other. XMI is an extension of the XML architecture. The XMI documents inherit the problems that arise with XML. The SRML description did not have a way to describe rules for the XMI document architecture. We have extended the SRML metalanguage to provide an effective method for describing calculation rules in an XMI environment making the compaction of XMI documents possible. Using this extension the tool we have implemented for XML compaction[10] can be used with minor modifications to enable XMI compaction.

In the future we plan to modify our existing XML compressor using this extension to create a universal XML/XMI compacting tool. The idea to create a demand-driven XMI compactor library is also planned, which would make the method a very effective tool in every day use, since the size decrease would mean that larger amount of information could be stored without sacrificing disk space. It could be used for example as an aid for third party database engines utilizing the XMI/XML format.

References

[1] Psaila, G. and Crespi-Reghizzi, S, 1999. *Adding Semantics to XML*, In Proceedings of the Second Workshop on Attribute Grammars and their Applications, WAGA'99 Amsterdam, The Netherlands, pages 113-132

[2] Knuth, D.E., 1968. *Semantics of Context-Free Languages*, Mathematical Systems Theory Vol 2., pages 127-145

[3] Bray, T. and Paoli, J. and Sperberg-McQueen, C., 1998. *Extensible markup language*, XML 1.0 W3C recommendation, http://www.w3.org/TR/REC-xml

[4] Liefke, H. and Suciu, D., 2000. *XMill: an efficient compressor for XML data* In Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, pages 153-164 http://www.research.att.com/sw/tools/xmlill/

[5] Hirakawa M. and Tanaka T., 2002. *JSNP: a database of common gene variations in the Japanese population*, Nucleic Acids Research Vol. 30, pages 158-162, Oxford University Press

[6] Schafner, B., *Model XML DTDs with Rational Rose* http://builder.com.com/5100-31-5075476.html

[7] Cokus, M. and Winkowski D., 2002. *XML Sizing and Compression Study For Military Wireless Data*, XML Conference and Exposition, Baltimore Convention Center, Baltimore, MD

[8] Ferenc, R. and Beszédes, Á. et al, 2002., *Columbus – Reverse Engineering Tool and Schema for C++*, In Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002), IEEE Computer Society, pages 172-181, Montréal, Canada

[9] Alblas, H., 1991. *Introduction to Attribute Gammars*, In Proceedings of SAGA LNCS Vol 545.,Springer-Verlag, pages 1-16

[10] Kálmán, M., Havasi, F., Gyimóthy, T., 2005. *Compacting XML documents*, Accepted for publication in The Journal of Information and Software Technology, Elsevier.

[11] Havasi, F., 2002. *XML Semantics Extension*, Acta Cybernetica Vol 15 No. 2, pages 509-528

[12] *OMG Unified Modeling Language Specification, Version 2.0*, http://www.omg.org

[13] *XML Metadata Interchange (XMI) Version 1.1*, http://cgi.omg.org/docs/ad/99-10- 02.pdf

[14] *Unified Modeling Language (UML)*, http://www.uml.org/

[15] *Common Object Request Broker Architecture (CORBA)*, http://www.omg.org/gettingstarted/corbafaq.htm

[16] *Meta-Object Facility (MOF), version 1.4*, http://www.omg.org/technology/documents/formal/mof.htm

[17] Goldfarb, C.F. and Prescod, P. 2001. *The XML Handbook*, Prentice-Hall

Appendix

A Example for automobile storage in XMI

```
<!DOCTYPE XMI SYSTEM "auto.dtd">
<XMI xmi.version="1.0">
 <XMI.header>
   <XMI.documentation>Some automobiles.</XMI.documentation>
 </XMI.header>
   <XMI.content>
      <Auto>
         <Auto.Make>Ford</Auto.Make>
         <Auto.Model>Mondeo</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.Color>Black</Auto.Color>
         <Auto.NetPrice>25000</Auto.NetPrice>
         <Auto.Tax>15</Auto.Tax>
         <Auto.SalesPrice>28750</Auto.SalesPrice>
     </Auto>
     <Auto>
         <Auto.Make>Opel</Auto.Make>
         <Auto.Model>Astra</Auto.Model>
         <Auto.Year>2004</Auto.Year>
         <Auto.Color>Papyrus</Auto.Color>
         <Auto.NetPrice>20000</Auto.NetPrice>
         <Auto.Tax>15</Auto.Tax>
         <Auto.SalesPrice>23000</Auto.SalesPrice>
     </Auto>
     <Auto>
         <Auto.Make>Volvo</Auto.Make>
         <Auto.Model>S40</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.Color>Red</Auto.Color>
         <Auto.NetPrice>30000</Auto.NetPrice>
         <Auto.Tax>15</Auto.Tax>
         <Auto.SalesPrice>34500</Auto.SalesPrice>
     </Auto>
     <Auto>
         <Auto.Make>Fiat</Auto.Make>
         <Auto.Model>Stilo</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.Color>Red</Auto.Color>
         <Auto.NetPrice>18000</Auto.NetPrice>
         <Auto.Tax>15</Auto.Tax>
         <Auto.SalesPrice>20700</Auto.SalesPrice>
     </Auto>
     <Auto>
         <Auto.Make>Toyota</Auto.Make>
         <Auto.Model>Corolla</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.Color>Red</Auto.Color>
         <Auto.NetPrice>24000</Auto.NetPrice>
         <Auto.Tax>15</Auto.Tax>
         <Auto.SalesPrice>27600</Auto.SalesPrice>
     </Auto>
```

```
   </XMI.content>
 </XMI.header>
</XMI>
```

B The SRML for the automobile example

```
<SRML>
 <rules-for root="Auto">
  <rule element="Auto" attrib="SalesPrice" reftype="text">
   <expr>
   <binary-op op="mul">
    <expr>
     <binary-op op="add">
       <expr>
         <binary-op op="div">
          <expr><attribute element="Auto" attrib="Tax" reftype="text"/></expr>
          <expr><data>100</data></expr>
         </binary-op>
        </expr>
       <expr><data>1</data></expr>
     </binary-op>
    </expr>
    <expr><attribute element="Auto" attrib="NetPrice" reftype="text"/></expr>
   </expr>
   </binary-op>
   </expr>
  </rule>
  <rule element="Auto" attrib="Tax" reftype="text">
   <expr>
    <data>15</data>
   </expr>
  </rule>
  <rule element="Auto" attrib="Color" reftype="text">
   <expr>
    <if-expr>
     <expr>
      <binary-op op="equals">
       <expr><attribute element="Auto" attrib="Year" reftype="text"/></expr>
       <expr><data>2000</data></expr>
      </binary-op>
     </expr>
     <expr><data>Red</data></expr>
     <expr><no-data/></expr>
    </if-expr>
   </expr>
  </rule>
 </rules-for>
</SRML>
```

C The compacted XMI example

```
<!DOCTYPE XMI SYSTEM "auto.dtd">
<XMI xmi.version="1.0">
 <XMI.header>
   <XMI.documentation>Some automobiles.</XMI.documentation>
 </XMI.header>
   <XMI.content>
      <Auto>
         <Auto.Make>Ford</Auto.Make>
         <Auto.Model>Mondeo</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.Color>Black</Auto.Color>
         <Auto.NetPrice>25000</Auto.NetPrice>
     </Auto>
     <Auto>
         <Auto.Make>Opel</Auto.Make>
         <Auto.Model>Astra</Auto.Model>
         <Auto.Year>2004</Auto.Year>
         <Auto.Color>Papyrus</Auto.Color>
         <Auto.NetPrice>20000</Auto.NetPrice>
     </Auto>
     <Auto>
         <Auto.Make>Volvo</Auto.Make>
         <Auto.Model>S40</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.NetPrice>30000</Auto.NetPrice>
     </Auto>
     <Auto>
         <Auto.Make>Fiat</Auto.Make>
         <Auto.Model>Stilo</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.NetPrice>18000</Auto.NetPrice>
     </Auto>
     <Auto>
         <Auto.Make>Toyota</Auto.Make>
         <Auto.Model>Corolla</Auto.Model>
         <Auto.Year>2000</Auto.Year>
         <Auto.NetPrice>24000</Auto.NetPrice>
     </Auto>
   </XMI.content>
 </XMI.header>
</XMI>
```

D An XMI output for the CPP example

```
...
<!-- ====================      Math  [class]       ==================== -->
  <Foundation.Core.Class xmi.id = 'id101'>
    <Foundation.Core.ModelElement.name>Math</Foundation.Core.ModelElement.name>
    <Foundation.Core.ModelElement.visibility xmi.value = 'public' />
    <Foundation.Core.ModelElement.isSpecification xmi.value = 'true'/>
    <Foundation.Core.GeneralizableElement.isRoot xmi.value = 'true' />
    <Foundation.Core.GeneralizableElement.isLeaf xmi.value = 'true' />
    <Foundation.Core.GeneralizableElement.isAbstract xmi.value = 'false' />
```

```
<Foundation.Core.Class.isActive xmi.value = 'false' />
<Foundation.Core.ModelElement.namespace>
  <Model_Management.Package xmi.idref = 'id100' /> <!-- global namespace -->
</Foundation.Core.ModelElement.namespace>
  <Foundation.Core.Classifier.feature>
  <!-- ============ Math [class] result [Attribute] =========== -->
  <Foundation.Core.Attribute xmi.id = 'id102' >
    <Foundation.Core.ModelElement.name>result
    </Foundation.Core.ModelElement.name>
    <Foundation.Core.ModelElement.visibility xmi.value = 'public' />
    <Foundation.Core.Feature.ownerScope xmi.value = 'classifier'/>
    <Foundation.Core.StructuralFeature.type>
      <Foundation.Core.DataType xmi.idref = 'id_dt_1'/>    <!-- float -->
    </Foundation.Core.StructuralFeature.type>
  </Foundation.Core.Attribute>
  </Foundation.Core.Classifier.feature>
  <Foundation.Core.Classifier.feature>
  <!-- ========== Math [class] GetResult [Operation] =========== -->
  <Foundation.Core.Operation xmi.id = 'id105' >
    <Foundation.Core.ModelElement.name>GetResult
    </Foundation.Core.ModelElement.name>
    <Foundation.Core.ModelElement.visibility xmi.value = 'public' />
    <Foundation.Core.BehavioralFeature.parameter>
      <Foundation.Core.Parameter xmi.id = 'id105.Return' >
        <Foundation.Core.ModelElement.name>GetResult.Return
        </Foundation.Core.ModelElement.name>
        <Foundation.Core.Parameter.kind xmi.value = 'return'/>
        <Foundation.Core.Parameter.type>
          <Foundation.Core.DataType xmi.idref = 'id_dt_1' />    <!-- float -->
        </Foundation.Core.Parameter.type>
      </Foundation.Core.Parameter>
    </Foundation.Core.BehavioralFeature.parameter>
  </Foundation.Core.Operation>
  </Foundation.Core.Classifier.feature>
  <Foundation.Core.Classifier.feature>
  <!-- ============= Math [class] Add [Operation] ============ -->
  <Foundation.Core.Operation xmi.id = 'id111' >
    <Foundation.Core.ModelElement.name>Add</Foundation.Core.ModelElement.name>
    <Foundation.Core.ModelElement.visibility xmi.value = 'public' />
    <Foundation.Core.BehavioralFeature.parameter>
      <Foundation.Core.Parameter xmi.id = 'id111.Return' >
        <Foundation.Core.ModelElement.name>Add.Return
        </Foundation.Core.ModelElement.name>
        <Foundation.Core.Parameter.kind xmi.value = 'return'/>
        <Foundation.Core.Parameter.type>
          <Foundation.Core.DataType xmi.idref = 'id_dt_1' />    <!-- float -->
        </Foundation.Core.Parameter.type>
      </Foundation.Core.Parameter>
        <Foundation.Core.Parameter xmi.id = 'id112' >
          <Foundation.Core.ModelElement.name>a
          </Foundation.Core.ModelElement.name>
          <Foundation.Core.ModelElement.visibility xmi.value = 'public'/>
          <Foundation.Core.Parameter.kind xmi.value = 'inout'/>
          <Foundation.Core.Parameter.type>
            <Foundation.Core.DataType xmi.idref = 'id_dt_1' />    <!-- float -->
          </Foundation.Core.Parameter.type>
```

```
            </Foundation.Core.Parameter>
            <Foundation.Core.Parameter xmi.id = 'id113' >
              <Foundation.Core.ModelElement.name>b
              </Foundation.Core.ModelElement.name>
              <Foundation.Core.ModelElement.visibility xmi.value = 'public'/>
              <Foundation.Core.Parameter.kind xmi.value = 'inout'/>
              <Foundation.Core.Parameter.type>
                <Foundation.Core.DataType xmi.idref = 'id_dt_1' />    <!-- float -->
              </Foundation.Core.Parameter.type>
            </Foundation.Core.Parameter>
          </Foundation.Core.BehavioralFeature.parameter>
        </Foundation.Core.Operation>
        </Foundation.Core.Classifier.feature>
        <Foundation.Core.Classifier.feature>
        ...
```

E The SRML of the XMI form of the CPP program

```
<SRML>
 <rules-for root="Foundation.Core.ModelElement.visibility>
  <rule element="srml:root" attrib="xmi.value">
   <expr>
    <data>public</data>
   </expr>
  </rule>
 </rules-for>
 <rules-for root="Foundation.Core.DataType">
  <rule element="srml:root" attrib="xmi.idref">
   <expr>
    <data>id_dt_1</data>
   </expr>
  </rule>
 </rules-for>
 <rules-for root="Foundation.Core.Parameter">
  <rule element="Foundation.Core.Parameter.kind" attrib="xmi.value">
   <epxr>
    <if-expr>
     <epxr>
      <binary-op op="ends-with">
       <expr>
        <attribute element="Foundation.Core.ModelElement.name" reftype="text">
       </expr>
       <expr><data>.Return</data></expr>
      </binary-op>
     </expr>
     <expr><data>return</data></expr>
     <expr><data>inout</data></expr>
    </if-expr>
   </expr>
  </rule>
 </rules-for>
</SRML>
```

F The compacted XMI of the CPP program

```
<!-- ====================     Math  [class]      ==================== -->
<Foundation.Core.Class xmi.id = 'id101' xmi.value='c:\\temp\\Math.cpp'>
  <Foundation.Core.ModelElement.name>Math</Foundation.Core.ModelElement.name>
  <Foundation.Core.ModelElement.visibility/>
  <Foundation.Core.ModelElement.isSpecification xmi.value = 'true'/>
  <Foundation.Core.GeneralizableElement.isRoot xmi.value = 'true' />
  <Foundation.Core.GeneralizableElement.isLeaf xmi.value = 'true' />
  <Foundation.Core.GeneralizableElement.isAbstract xmi.value = 'false' />
  <Foundation.Core.Class.isActive xmi.value = 'false' />
  <Foundation.Core.ModelElement.namespace>
    <Model_Management.Package xmi.idref = 'id100' /> <!--  global namespace  -->
  </Foundation.Core.ModelElement.namespace>
    <Foundation.Core.Classifier.feature>
    <!-- ==========     Math  [class]  result  [Attribute]     ============= -->
    <Foundation.Core.Attribute xmi.id = 'id102' >
      <Foundation.Core.ModelElement.name>result
      </Foundation.Core.ModelElement.name>
      <Foundation.Core.ModelElement.visibility/>
      <Foundation.Core.Feature.ownerScope xmi.value = 'classifier'/>
      <Foundation.Core.StructuralFeature.type>
        <Foundation.Core.DataType/>    <!-- float -->
      </Foundation.Core.StructuralFeature.type>
    </Foundation.Core.Attribute>
    </Foundation.Core.Classifier.feature>
    <Foundation.Core.Classifier.feature>
    <!-- =========     Math  [class]  GetResult  [Operation]     ========== -->
    <Foundation.Core.Operation xmi.id = 'id105'>
      <Foundation.Core.ModelElement.name>GetResult
      </Foundation.Core.ModelElement.name>
      <Foundation.Core.ModelElement.visibility/>
      <Foundation.Core.BehavioralFeature.parameter>
        <Foundation.Core.Parameter xmi.id = 'id105.Return' >
          <Foundation.Core.ModelElement.name>GetResult.Return
          </Foundation.Core.ModelElement.name>
          <Foundation.Core.Parameter.kind/>
          <Foundation.Core.Parameter.type>
            <Foundation.Core.DataType/>    <!-- float -->
          </Foundation.Core.Parameter.type>
        </Foundation.Core.Parameter>
      </Foundation.Core.BehavioralFeature.parameter>
    </Foundation.Core.Operation>
    </Foundation.Core.Classifier.feature>
    <Foundation.Core.Classifier.feature>
    <!-- ============     Math  [class]  Add  [Operation]     ============= -->
    <Foundation.Core.Operation xmi.id = 'id111'>
      <Foundation.Core.ModelElement.name>Add</Foundation.Core.ModelElement.name>
      <Foundation.Core.ModelElement.visibility/>
      <Foundation.Core.BehavioralFeature.parameter>
        <Foundation.Core.Parameter xmi.id = 'id111.Return' >
          <Foundation.Core.ModelElement.name>Add.Return
          </Foundation.Core.ModelElement.name>
          <Foundation.Core.Parameter.kind/>
          <Foundation.Core.Parameter.type>
            <Foundation.Core.DataType/>    <!-- float -->
```

```
          </Foundation.Core.Parameter.type>
        </Foundation.Core.Parameter>
          <Foundation.Core.Parameter xmi.id = 'id112' >
            <Foundation.Core.ModelElement.name>a
            </Foundation.Core.ModelElement.name>
            <Foundation.Core.ModelElement.visibility/>
            <Foundation.Core.Parameter.kind/>
            <Foundation.Core.Parameter.type>
              <Foundation.Core.DataType/>    <!-- float -->
            </Foundation.Core.Parameter.type>
          </Foundation.Core.Parameter>
          <Foundation.Core.Parameter xmi.id = 'id113' >
            <Foundation.Core.ModelElement.name>b
            </Foundation.Core.ModelElement.name>
            <Foundation.Core.ModelElement.visibility/>
            <Foundation.Core.Parameter.kind/>
            <Foundation.Core.Parameter.type>
              <Foundation.Core.DataType/>    <!-- float -->
            </Foundation.Core.Parameter.type>
          </Foundation.Core.Parameter>
      </Foundation.Core.BehavioralFeature.parameter>
    </Foundation.Core.Operation>
    </Foundation.Core.Classifier.feature>
    <Foundation.Core.Classifier.feature>
    ...
```

G The DTD of the extended SRML metalanguage

```
<!ELEMENT semantic-rules (rules-for*)>
<!ELEMENT rules-for(rule*)>
<!ATTLIST rules-for root NMTOKEN #REQUIRED >
<!ELEMENT rule(expr)>
<!ATTLIST rule element NMTOKEN #REQUIRED
              attrib NMTOKEN #REQUIRED
              reftype (text|attrib) "attrib">
<!ELEMENT expr (binary-op | attribute | data
               | no-data | if-element
               | if-expr | if-all | if-any
               | current-attribute | position)>
<!ELEMENT binary-op (expr, expr)>
<!ATTLIST binary-op
       op (add | sub | mul | div | exp | equal
               | not-equal |less | greater | or
               | xor | and | nor | contains
               | concat | begins-with
               | ends-with) #REQUIRED >
<!ELEMENT position EMPTY>
<!ATTLIST position  element NMTOKEN "srml:all"
                    from (begin | current
                                | end) "begin">
<!ELEMENT attribute EMPTY>
<!ATTLIST attribute element NMTOKEN "srml:this"
                    num NMTOKEN "0"
                    from (begin | current
                                | end) "current"
```

```
                    type (temp | permanent) "permanent"
                    attrib NMTOKEN #REQUIRED
                    reftype (text|attrib) "attrib">
<!ELEMENT if-element (expr, expr)>
<!ATTLIST if-element from(begin | end) "begin">
<!ELEMENT if-all (expr, expr, expr)>
<!-- cond,if,else-->
<!ATTLIST if-all element NMTOKEN "srml:all"
                  attrib NMTOKEN "srml:all"
                  reftype (text|attrib) "attrib">
<!ELEMENT if-any (expr, expr, expr)>
<!--cond,if,else-->
<!ATTLIST if-any element NMTOKEN "srml:all"
                  attrib NMTOKEN "srml:all"
                  reftype (text|attrib) "attrib">
<!ELEMENT current-attribute EMPTY>
<!ELEMENT if-expr (expr,expr,expr)>
<!-- condition , if, else -->
<!ELEMENT data (#PCDATA)>
<!ELEMENT no-data EMPTY>
<!ELEMENT extern-function (param)*>
<!ATTLIST extern-function name NMTOKEN #REQUIRED>
<!ELEMENT param(expr)>
```

Acta Cybernetica 17 (2005) 311–323.

Classification using a sparse combination of basis functions

Kornél Kovács* and András Kocsor*

Abstract

Combinations of basis functions are applied here to generate and solve a convex reformulation of several well-known machine learning algorithms like certain variants of boosting methods and Support Vector Machines. We call such a reformulation a Convex Networks (CN) approach. The nonlinear Gauss-Seidel iteration process for solving the CN problem converges globally and fast as we prove. A major property of CN solution is the sparsity, the number of basis functions with nonzero coefficients. The sparsity of the method can effectively be controlled by heuristics where our techniques are inspired by the methods from linear algebra. Numerical results and comparisons demonstrate the effectiveness of the proposed methods on publicly available datasets. As a consequence, the CN approach can perform learning tasks using far fewer basis functions and generate sparse solutions.

1 Introduction

Numerous scientific areas such as optical character and speech recognition, speaker verification, bioinformatics and pharmacology nowadays significantly depend on statistical machine learning algorithms of artificial intelligence. The common feature of these areas - artificial knowledge embedded in applications - is retrieved from pre-collected databases in a statistical way. Recently the size of the data sets for calibrating the methods has grown due to advances in global communication networks like the Internet. Processing this extra amount of data requires effective methods that store the extracted information in a compact and easily retrievable form.

One of the most prevalent machine learning algorithms - Artificial Neural Networks (ANN) [3] - meets these requirements as it has compact form with a fast evaluation. However the solution provided by the learning phase is only a local minima of the objective function, which makes the networks trained on the same database inconsistent. The ubiquitous Support Vector Machine (SVM) method [6, 9, 18] leads to a quadratic programming task whose own global optima defines

*Research Group on Artificial Intelligence of the Hungarian Academy of Sciences, H-6720 Szeged, Aradi vértanúk tere 1., Hungary, e-mail: {kkornel,kocsor}@inf.u-szeged.hu

the compactness of the information retrieved. This kind of functioning can be beneficial since preliminary assumptions are not required, but this is also why the technique might not be applicable in every case. Our aim is to define an algorithm which combines the advantages of the methods and, in particular, it has global optima even with controlled sparsity.

Now we will briefly outline the contents of the paper. First we state the pattern classification problem and derive the so called Convex Networks (CN) method from a constrained optimization formulation in Eq. (8). The nonlinear Gauss-Seidel iteration technique in Definition 2 for solving the CN problem converges globally as shown in the Optimization section without proof. To demonstrate CN's flexibility the original SVM quadratic programming task is re-expressed in a CN form. In the next section we introduce heuristics for controlling the sparsity of the solution. In the numerical tests and comparisons section we demonstrate the practical applicability of CN compared with ANN and SVM. Lastly, we round off with our conclusions and some ideas for future research.

2 Convex networks

Tasks in machine learning often lead to classification and regression problems where models employing a convex objective function might be beneficial. Consider the problem of classifying n points in a compact set $\mathcal{X}$ over $\mathbb{R}^m$, represented by $\mathbf{x}_1, \ldots, \mathbf{x}_n$, according to the membership of each point $\mathbf{x}_i$ in the classes $\{1, \ldots, c\}$ as specified by $y_1, \ldots, y_n$. A multiclass problem can be transformed into a set of binary classification tasks $y_i \in \{-1, +1\}$, which is in many ways like the one-against-all method [20] or the output coding scheme [13]. Thus our investigation can be restricted to the problem of the binary classification without any loss of generality.

Solutions to classification problems in practice are usually based on the model-method where the parameters of a fixed model structure are set by statistics-based optimization. The structure can depend on compact mathematical models [3, 18] or it could apply the points themselves of the available database [8]. Models accomplish the separation by estimating the probability density functions of different classes [1], or by utilizing a separator surface between the points. In both cases we need to look for models which return the following probabilities:

$$P(y \mid \mathbf{z}) \qquad y \in \{-1, +1\}, \quad \mathbf{z} \in \mathcal{X}. \tag{1}$$

The latter case is the discriminative approach where the separator surface is defined by the following set for a fixed $\gamma \in \mathbb{R}$

$$\{\mathbf{z} \mid f(\mathbf{z}) = \gamma,\ \mathbf{z} \in \mathcal{X}\}, \qquad f : \mathcal{X} \to \mathbb{R}. \tag{2}$$

The classification of an arbitrary point $\mathbf{z}$ is based on the sign of $f(\mathbf{z}) - \gamma$, and the probabilities in Eq. (1) could be derived by taking the amplitude of this quantity.

Now let S denote a finite set of continuous basis functions

$$S = \{f_1(\mathbf{x}), \ldots, f_k(\mathbf{x})\}, \qquad f_i : \mathcal{X} \to \mathbb{R} \tag{3}$$

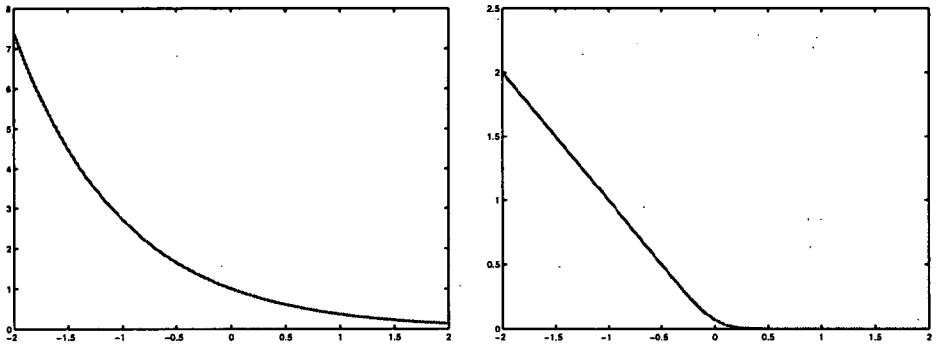


Figure 1: Possible loss functions

and the optimal separator surface of discriminative approach in Eq. (2) is searched for in the linear subspace of basis functions, $f \in Span(S)$, where

$$Span(S) = \left\{ h : \mathcal{X} \to \mathbb{R} \mid h(\mathbf{x}) = \sum_{i=1}^{k} \alpha_i f_i(\mathbf{x}),\ \mathbf{x} \in \mathcal{X},\ \boldsymbol{\alpha} \in \mathbb{R}^k \right\}. \tag{4}$$

Generally the optimality criterion is based on a special indicator of the sample points

$$y_i f(\mathbf{x}_i) \qquad 1 \leq i \leq n, \tag{5}$$

whose amplitudes are proportional to point-surface distances, positive values representing the well separated cases. Recalling that separable classification problems have an infinite number of separator surfaces that can classify the sample points perfectly, we introduce a twice continuously differentiable, monotone decreasing, lower bounded and convex loss-function $L : \mathbb{R} \to \mathbb{R}$ [9]. Of the many possibilities two candidates are shown in Fig. 1. Using a loss-function the separation measure $g(\boldsymbol{\alpha})$ can be defined for a function $f \in Span(S)$ and samples $\mathbf{x}_1, \ldots, \mathbf{x}_n$ by

$$g(\boldsymbol{\alpha}) = \sum_{i=1}^{n} L\left(y_i \sum_{j=1}^{k} \alpha_j f_j(\mathbf{x}_i) \right). \tag{6}$$

2.1 Optimization methods

A machine learning method can be regarded as a multivariate regression problem where the probabilities in Eq. (1) need to be approximated. The parameters of the applied model can be optimally set only if the estimated function is known over the whole space. The problem of approximating the parameters based on sparse sample data is ill-conditioned and the classical way of solving it is to use

regularization theory [17]. According to this theory the optimal separator surface has the minimal separation measure of Eq. (6) with a regularization term

$$\begin{array}{ll} \min_{\alpha} & \sum_{i=1}^{n} L\left(y_i \sum_{j=1}^{k} \alpha_j f_j(\mathbf{x}_i)\right) + \lambda \alpha^T A \alpha \\ \text{s.t.} & \alpha \in \mathbb{R}^k \end{array} \tag{7}$$

where $\lambda > 0$ and $A \in \mathbb{R}^{k \times k}$ is an arbitrary symmetric positive-definite matrix.

In practical applications constraints can be employed on the subspace of basis functions in the form of $\alpha \in \mathcal{A} \subseteq \mathbb{R}^k$ where $\mathcal{A}$ is a non-empty, closed, convex set. We will restrict our investigation here to the case where the domain is a product of non-empty intervals, i.e. $\mathcal{A} = \mathcal{A}_1 \times \ldots \times \mathcal{A}_k$. The formalism includes the unconstrained task of Eq. (7) where $\mathcal{A}_i = (-\infty, \infty)$. The final form of the Convex Networks (CN) problem is

$$\begin{array}{ll} \min_{\alpha} & \sum_{i=1}^{n} L\left(y_i \sum_{j=1}^{k} \alpha_j f_j(\mathbf{x}_i)\right) + \lambda \alpha^T A \alpha \\ \text{s.t.} & \alpha \in \mathcal{A} = \mathcal{A}_1 \times \ldots \times \mathcal{A}_k \end{array} \tag{8}$$

It can be readily seen that the objective function in this equation is twice continuously differentiable, lower bounded and convex. Moreover, every level set is bounded. Actually, Eq. (8) is a convex programming task which can be solved by one of many techniques [2].

The Sequential Quadratic Programming (SQP) methods [7, 10] focus on the solving of Kuhn-Tucker (KT) equations, which are sufficient conditions for global optima in the convex programming case. SQP is an iterative algorithm for solving a quadratic programming subproblem at each step. The convergence of SQP is super-linear due to the special update rule of second order information about KT equations.

In contrast to SQP, the Gauss-Seidel (GS) iteration technique is a kind of convergent algorithm that modifies one component of the solution at each step - in other words a simple convex optimization subproblem with one variable is solved at each step. Hence the resource requirements of the method remain bounded even for large-sized datasets. That is why we prefer to use the GS method to solve a CN task.

Definition 1. (projection mapping)

$$[\,]^p : \mathbb{R}^k \to \mathcal{A} \qquad [\alpha]^p = \mathbf{z} \Leftrightarrow \|\alpha - \mathbf{z}\|^2 = \min_{\mathbf{y} \in \mathcal{A}} \|\alpha - \mathbf{y}\|^2$$

Definition 2. (constrained Gauss-Seidel iteration)

$$\alpha_i^{t+1} = \left[\alpha_i^t - \gamma \nabla_i f(\mathbf{z}_i^t)\right]_i^p$$

where

$$\gamma > 0, \qquad \mathbf{z}_i^t = (\alpha_1^{t+1}, \ldots, \alpha_{i-1}^{t+1}, \alpha_i^t, \ldots, \alpha_k^t), \qquad \alpha^{t+1} = \mathbf{z}_{k+1}^t.$$

During the iteration process each component of the actual solution $\boldsymbol{\alpha}^t$ is successively upgraded by the gradient rule. If the solution falls outside the domain it will be replaced by the nearest point of the set with the aid of the projection mapping. The constrained GS iteration method is convergent for every function $\tau : \mathcal{A} \to \mathbb{R}$ over a non-empty, convex and closed set $\mathcal{A}$, where τ is twice continuously differentiable and lower bounded. Moreover, the gradient should be a Lipschitz function and there must exist a $\delta > 0$ such that $0 < \delta \leq \nabla^2_{ii}\tau(\boldsymbol{\alpha}))$. The limit point of the iteration is the extreme of the function over $\mathcal{A}$ [2].

However it can be proved that the Lipschitz condition respecting the gradient can be ignored if every level set of the function is bounded. Therefore the constrained Gauss-Seidel iteration procedure with low resource requirements is proposed for solving the CN task.

2.2 Methods involved

The CN formalism includes several well-known machine learning algorithms e.g. variants of boosting methods [11, 12] and Support Vector Machines (SVM) [6, 14, 16].

The standard SVM problem is given by the following for some $C > 0$, taking into account the fact that the bias in the separator hyperplane may be eliminated from the equation [15]:

$$\begin{array}{rlcl} \min_{\mathbf{w}} & C\mathbf{e}^T\boldsymbol{\xi} + \frac{1}{2}\mathbf{w}^T\mathbf{w} & & \\ \text{s.t.} & YX\mathbf{w} + \boldsymbol{\xi} & \geq & \mathbf{e}\ , \\ & \boldsymbol{\xi} & \geq & \mathbf{0} \end{array} \tag{9}$$

where Y is a diagonal matrix with $y_1, \ldots, y_n$ along its diagonal, $X = (\mathbf{x}_1, \ldots, \mathbf{x}_n)^T$ and $\mathbf{e}$ is a column vector of ones of arbitrary dimension. To solve this optimization problem we have to find the saddle point of the Lagrangian

$$\begin{array}{rl} \max_{\mathbf{w},\boldsymbol{\alpha}} & C\mathbf{e}^T\boldsymbol{\xi} + \frac{1}{2}\mathbf{w}^T\mathbf{w} - \boldsymbol{\alpha}^T(YX\mathbf{w} + \boldsymbol{\xi} - \mathbf{e}) \\ \text{s.t.} & \boldsymbol{\alpha}, \boldsymbol{\xi} \geq \mathbf{0} \end{array} \tag{10}$$

The parameters that maximize the Lagrangian must satisfy the conditions

$$\mathbf{w} = X^TY\boldsymbol{\alpha} \qquad \mathbf{0} \leq \boldsymbol{\alpha} \leq C\mathbf{e}. \tag{11}$$

These set of constraints can be employed in the original problem of Eq. (9) because the duality gap disappears when the objective function is convex

$$\begin{array}{rlcl} \min_{\boldsymbol{\alpha}} & C\mathbf{e}^T\boldsymbol{\xi} + \frac{1}{2}\boldsymbol{\alpha}^TYKY\boldsymbol{\alpha} & & \\ \text{s.t.} & YKY\boldsymbol{\alpha} + \boldsymbol{\xi} & \geq & \mathbf{e} \\ & \boldsymbol{\alpha} & \geq & \mathbf{0}\ , \\ & -\boldsymbol{\alpha} & \geq & C\mathbf{e} \\ & \boldsymbol{\xi} & \geq & \mathbf{0} \end{array} \tag{12}$$

where $K_{ij} = \kappa(\mathbf{x}_i, \mathbf{x}_j)$ is the kernel matrix of the sample. Mapping $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a Mercer-kernel [6] which can define some implicit nonlinear transformation of

the original points so that $K = XX^T$ means a linear mapping. For a solution $\boldsymbol{\alpha}$ of Eq. (12), $\boldsymbol{\xi}$ is given by $(\mathbf{e} - YKY\boldsymbol{\alpha})_+$ where

$$(\mathbf{z}_+)_i = \max\{0, z_i\} \qquad i = 1, \ldots, n \tag{13}$$

Exploiting this in Eq. (12) we get

$$\begin{array}{ll} \min_{\boldsymbol{\alpha}} & \sum_{i=1}^{n} \left(1 - y_i \sum_{j=1}^{n} \alpha_j y_j K_{ij}\right)_+ + \frac{1}{2C}\boldsymbol{\alpha}^T YKY\boldsymbol{\alpha} \\ \text{s.t.} & \mathbf{0} \leq \boldsymbol{\alpha} \leq C\mathbf{e} \end{array} \tag{14}$$

which is a CN problem with the following parameters

$$\begin{array}{lll} k = n & L(x) = (1-x)_+ & f_j(\mathbf{z}) = y_j \kappa(\mathbf{z}, \mathbf{x}_j) \\ \lambda = \frac{1}{2C} & A = YKY & \mathcal{A} = [0, C]^n \end{array} \tag{15}$$

if the plus function $(1-x)_+$ is replaced by a very accurate smooth approximation $p(x) = -(1-x) + \frac{1}{\beta}\log(1 + e^{\beta(1-x)})$, $\beta \to \infty$. Actually, it can be shown that as the smoothing parameter β tends to infinity the unique solution of the smoothed problem approaches the unique solution of the equivalent task in Eq. (15) [14].

3 Sparse solutions

The separator surface coded by a CN problem takes the form

$$\{\mathbf{z} \mid \sum_{j=1}^{k} \alpha_j f_j(\mathbf{z}) = \gamma,\ \mathbf{z} \in \mathcal{X}\}, \qquad f_j : \mathcal{X} \to \mathbb{R}. \tag{16}$$

for a fixed threshold $\gamma \in \mathbb{R}$. Basis functions with zero coefficients can be eliminated when evaluating the model and the remaining terms define the complexity of the CN solution. The more the number of zero coefficients the faster the evaluation, which makes the CN method suitable for fast or real-time applications. However the coefficients are determined by the optimal solution of the mathematical programming task, and the parameters can only influence the sparsity by degrading the performance.

For the sake of controlling the complexity the number of basis functions will be restricted by making the following assumption on the CN domain

$$\sum_{i=1}^{k} |sign(\alpha_i)| \leq q \tag{17}$$

Such a condition violates the closed and convex properties of the domain so the suggested nonlinear Gauss-Seidel technique and other iterative methods cannot be applied to the problem. The last remaining approach is the combinatorial selection of basis functions. Our aim is to select from the available basis functions a subset of order q where the classification problem can be optimally solved. This task is NP hard so the only effective way here is to employ heuristics which can be based on the execution of CN with different parameters or their own objective functions. In the next part we will outline methods from the latter group.

3.1 Heuristics

In this section we deal with algorithms that do not use the CN objective function itself during the optimal basis function subset selection of order q.

`RANDOM` The simplest strategy is the random selection approach when we randomly select q basis functions from among the k basis functions. This approach does not have an objective function that can be minimized so we will choose instead the subset with the best performance after several executions.

`MGRAMM` The CN method approximates the optimal separator surface using a linear combination of the basis functions. Hence the approximation can be performed on an orthogonal basis of the function space, as in the case of the result of the Gramm-Schmidt orthogonalization algorithm. Despite this, the dimension of the basis is the rank of the function set which can exceed the desired number q. Moreover, the algorithm generates an orthogonal function system with linear combinations of basis functions instead of selecting the individual functions.

To solve the above we will define a greedy iterative selection strategy based on a modified version of the Gramm-Schmidt orthogonalization algorithm. Among the available basis functions we choose the one with a maximal residual norm after the Gramm-Schmidt process at each step. The result of this greedy method is not the orthogonal function system itself but the basis functions used in the linear combinations.

```
GRAMM(q)
  Y = {1,...,k};  I = ∅;
  for i = 1...q
    t = argmax_{j∈Y−I} ‖f_j − Σ_{l=1}^{i−1} (⟨f_j,f_l^*⟩ / ⟨f_l^*,f_l^*⟩) f_l^*‖_2
    I = I ∪ {t};
    f_i^* = f_t − Σ_{l=1}^{i−1} (⟨f_t,f_l^*⟩ / ⟨f_l^*,f_l^*⟩) f_l^*;
  return I;
```

Assume that the basis functions are elements of L_2 so the dot product is the integral of the product function. When analytical computations of the integrals are not possible we utilize the following approximation in the algorithm using the sample points

$$\langle f, g \rangle = \sum_{i=1}^{n} f(\mathbf{x}_i) g(\mathbf{x}_i) \qquad f, g : \mathcal{X} \to \mathbb{R}. \tag{18}$$

`CORR` The `MGRAMM` method tries to choose an orthogonal basis of the functions with the help of the Gramm-Schmidt process. The choice might be good when the dot product of functions is available. Employing the approximation in Eq. (18) the result of the algorithm will be also just an approximation of the desired basis.

Such an estimation can be carried out in different ways. The orthogonality of the elements in the basis can be also employed, since the mutual correlation coefficients must be zero. Our aim is to select functions such that the squared sum of the element in the correlation matrix should be minimal. Similar to MGRAMM this method will be a greedy iterative process and also exploit the fact that the mutual correlation coefficient for normalized functions takes the form of Eq. (18).

```
CORR(q)
  Y = {1,...,k}; I = ∅;
  for i = 1...q
    t = argmin_{j∈Y−I} Σ_{l=1}^{i−1} ⟨f_j, f_l^*⟩^2 / ⟨f_j, f_j⟩
    I = I ∪ {t};
    f_i^* = f_t / ⟨f_t, f_t⟩^0.5;
  return I;
```

4 Results

We now demonstrate the effectiveness of the CN approach by comparing its results with other methods. In order to evaluate how well each algorithm classifies an unknown dataset, we performed a tenfold cross-validation on publicly available datasets from the UCI repository [4]. The performance of the CN method was compared with Artificial Neural Networks (ANN) and Support Vector Machines (SVM).

We applied a feed-forward neural network (MLP) with one hidden layer, where the number of hidden neurons was set at three times the class number. The back-propagation learning rule was applied for training. MLP was executed five times on each dataset and then we chose the parameter values which gave the best performance on training sample.

For an impartial comparison we employed our 1-norm SVM implementation where the bias term was absent [15]. Multiclass cases were handled by the one-against-all approach. Additionally, the cosine polynomial kernel we applied made the SVM method nonlinear

$$\kappa(\mathbf{x},\mathbf{y}) = \left(\frac{\mathbf{x}^T\mathbf{y}}{\|\mathbf{x}\|\,\|\mathbf{y}\|} + \sigma\right)^q, \qquad q \in \mathbb{N},\ \sigma \in \mathbb{R}_+ \tag{19}$$

with parameters $q = 3$ and $\sigma = 1$.

The basis functions for the CN problem were defined by the above kernel function using the sample points of a training set, as shown in Eq. (14). Thus

$$f_j(\mathbf{z}) = y_j \kappa(\mathbf{z},\mathbf{x}_j). \qquad j = 1,\ldots,n \tag{20}$$

The coefficients of the basis functions were not restricted in our tests, i.e. we used the domain $\mathcal{A} = (-\infty,\infty)^n$. In the regularization term of Eq. (8) we set the identity matrix equal to A with $\lambda = 1$.

	ANN	SVM	CN
balance	89.03	93.55	99.79
	86.35	90.63	**95.41**
bupa	72.01	81.73	80.69
	68.07	**74.39**	71.92
glass	84.24	99.79	100.0
	69.87	84.70	**86.23**
iono	93.35	99.40	99.94
	86.17	91.09	**92.41**
monks	90.64	97.50	99.05
	87.28	95.82	**96.51**
pima	78.68	82.49	80.55
	76.09	75.58	74.82
wdbc	98.71	99.47	100.0
	97.61	**97.62**	96.93
wpbc	85.71	98.47	99.04
	76.41	77.36	**79.63**

Table 1: Ten-fold cross-validation training and testing results on some UCI datasets using three different methods. ANN is a feed-forward neural network with one hidden layer where the number of hidden units was set at three times the class number. SVM used the cosine polynomial kernel defined in Eq. (19) with $q = 3$ and $\sigma = 1$ for nonlinearity. With the help of Eq. (14) the CN method applied the same basis functions.

It turned out that, on most of the datasets tested, the tenfold testing correctness of the CN problem was the highest for these methods. We summarize all these results in Table 1. It confirms that the CN classification method is indeed just as effective as the ubiquitous machine learning algorithms. Moreover, their performances were surpassed in many cases. It can be readily seen that the problem of overfitting the data was present more often in the methods with global optima. It might be explained with the locally optimal solution of the ANN method, which can be regarded as a kind of regularization. Similar results are expected when using sparse heuristics to solve a CN problem.

We also examined the performance of the proposed heuristics controlling CN sparsity. We compared the methods on the Iono database by examining the value of the CN objective function, regardless of how the methods worked. Now the RANDOM method chose its best from 5 executions. The results of the heuristics are shown in Fig. 2. We used the performance of the RANDOM method as a reference so the results of other algorithms are expressed in percentages. As the reader will notice the MGRAMM and the CORR approaches achieved similar results, and both of them outperformed the RANDOM method here. Despite the fact that these algorithms require computational time the selected basis perform better.

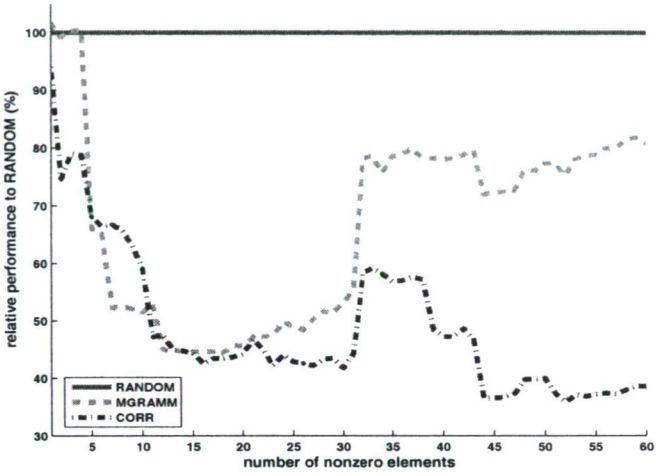


Figure 2: Performances of the proposed heuristics controlling CN sparsity on the Iono database expressed in percentages of the `RANDOM` method result. The CN measures were used as performance indicators regardless of how the methods works.

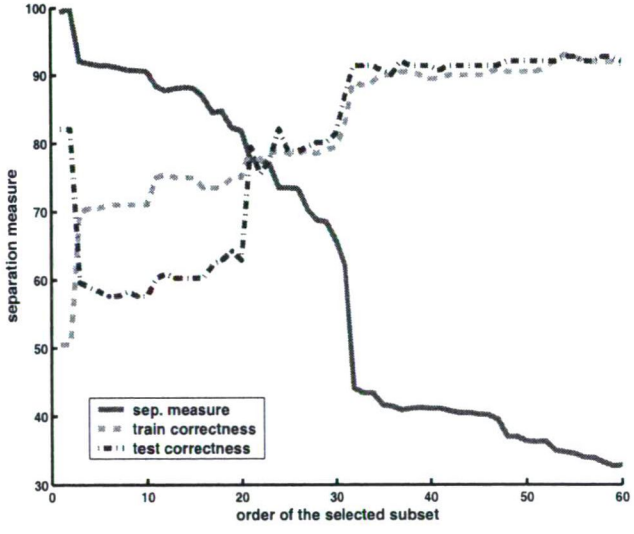


Figure 3: The consistency of the measure in the CN method and its abstraction ability with the aid of the `MGRAMM` method on the Iono database. The decreasing CN measure means a better testing correctness.

During the subset selection we optimize some measures while the abstraction ability is the most important in the machine learning sense. The consistency of the measure in the CN method and its abstraction ability can be seen in Fig. 3 with the aid of the MGRAMM method on the previous database. As can readily be seen, the decreasing CN measure value means a better abstraction ability, i.e. testing correctness. Thus the measure of the CN approach might indeed be employed as an objective function of machine learning algorithms.

The performance of heuristics were examined with the help of ten-fold cross-validation. We summarize our results here in Table 2. The sparsity of solutions were maximized using 10%, 20% and 30% of the available functions. The RANDOM

RANDOM MGRAMM CORR	10%	20%	30%	100%
	95.10	95.25	95.40	
balance	95.25	95.40	95.25	95.41
	95.41	95.10	95.25	
	70.49	71.35	69.14	
bupa	69.14	70.53	**71.61**	71.92
	69.12	69.12	69.42	
	84.75	**89.66**	87.00	
glass	85.16	86.62	85.91	86.23
	85.18	85.16	86.66	
	89.16	**93.19**	92.68	
iono	91.23	92.04	90.54	92.41
	91.58	91.32	91.85	
	93.18	93.70	94.76	
monks	92.94	91.66	90.89	96.51
	92.65	94.40	**95.11**	
	78.51	77.24	75.97	
pima	77.89	76.43	77.87	74.82
	77.62	76.22	76.60	
	97.44	97.27	97.44	
wdbc	97.25	96.93	96.09	96.93
	97.10	96.93	96.93	
	78.27	78.29	77.29	
wpbc	76.37	75.14	73.20	79.63
	74.05	75.93	**79.70**	

Table 2: Ten-fold cross-validation testing results of the Convex Networks method using the heuristics RANDOM, MGRAMM and CORR. The sparsity was controlled by maximizing the number of available basis functions to 10%, 20% and 30% of the complete sets, respectively.

method had the same parameter as that above. As observed, all of the algorithms selected subsets with adequate testing correctness. This kind of capacity reduction in the CN learning method brings about a sort of regularization which is reflected in the results: results with a reduced basis outperform the original ones in many cases. The various algorithms here have their best performance on different tasks. In general, different requirements in the learning phase will lead the user to select one of the available heuristics.

5 Conclusions

We proposed a reformulation of certain machine learning algorithms that includes several well-known nonlinear classification methods. The CN problem can be solved by the convergent nonlinear Gauss-Seidel iteration process, which is sufficiently fast for this task. The numerical results on its abstraction ability show that the CN method can be considered as a rival classification method to both ANN and SVM. Moreover, the sparsity of the CN problem can be effectively controlled by the proposed heuristics. Future work includes a new heuristic based on a CN objective function which can be utilized in very large classification problems. We also plan to use chunking algorithms like those described in [5] for problems which do not fit in the memory.

References

[1] Alder, M. D. *Principles of Pattern Classification: Statistical, Neural Net and Syntactic Methods of Getting Robots to See and Hear,* http://ciips.ee.uwa.edu.au/~mike/PatRec, 1994.

[2] Bertsekas, D.P. And Tsitsiklis, J. N. *Parallel and Distributed Computation: Numerical Methods,* Prentice Hall, 1989; republished by Athena Scientific, 1997.

[3] Bishop, C. M., *Neural Networks for Pattern Recognition,* Oxford University Press, 1995.

[4] Blake, C. L. and Merz, C. J. *UCI repository of machine learning databases,* http://www.ics.uci.edu/ mlearn/MLRepository.html, 1998.

[5] Bradley, P. S. and Mangasarian, O. L. *Massive data discrimination via linear support vector machines,* Optimization Methods and Softwares, vol. 13, pp. 1-10, 2000.

[6] Cristianini, N. And Shawe-Taylor, J. *An Introduction to Support Vector Machines and other kernel-based learning methods,* Cambridge University Press, 2000.

[7] Conn, A. R., Gould, N. I. M., Toint, T. L. *Trust-region methods,* Society for Industrial and Applied Mathematics, 2000.

[8] Duda, R. and Hart, P. *Pattern Classification and Scene Analysis,* Wiley and Sons, New York, 1973.

[9] Evgeniou, T., Pontil, M., Poggio, T. *Regularization Networks and Support Vector Machines,* Advances in Computational Mathematics, Vol. 13/1, pp. 1-50, 2000.

[10] Fletcher, R. *Practical Methods of Optimization,* John Wiley and Sons, 1987.

[11] Freund, Y. and Schapire, R.E. *A decision-theoretic generalization of on-line learning and an application to boosting,* J. Comput. Syst. Sci., vol. 55/1, pp. 119-139, 1997.

[12] Friedman, J., Hastie, T., Tibshirani, R. *Additive logistic regression: A statistical view of boosting,* The Annals of Statistics, vol. 28/2, pp. 337-407, 2000.

[13] Kong, E. B. and Dietterich, T. *Error-Correcting Output Coding Corrects Bias and Variance* International Conference on Machine Learning, pp. 313-321, 1995.

[14] Lee, Y.-J. and Mangasarian, O. L. *SSVM: A Smooth Support Vector Machine for Classification,* Computational Optimization and Applications, vol. 20/1, pp. 5-22, 2001.

[15] Poggio, T., Mukherjee, S., Rifkin, R., Rakhlin, A., Verri, A. *b,* in Proceedings of the Conference on Uncertainty in Geometric Computations, 2001.

[16] Suykens, J.A.K. and Vandewalle, J. *Least squares support vector machine classifiers,* Neural Processing Letters, 1999.

[17] Tikhonov, A. N. And Arsenin, V. Y. *Solutions of Ill-posed Problems,* W. H. Winston, Washington, D.C., 1977.

[18] Vapnik, V. N. *Statistical Learning Theory,* John Wiley & Sons Inc., 1998.

[19] Wahba, G. *Splines models for Observational Data,* Series in Applied Mathematics, Vol. 59, SIAM, Philadelphia, 1990.

[20] Weston, J. and Watkins, C. *Support vector machines for multiclass pattern recognition,* Proceedings of the Seventh European Symposium On Artificial Neural Networks, 1999.

Acta Cybernetica **17** (2005) 325–338.

HyperS Tableaux – Heuristic Hyper Tableaux

Gergely Kovásznai*

Abstract

Several syntactic methods have been constructed to automate theorem proving in first-order logic. The positive (negative) hyper-resolution and the clause tableaux were combined in a single calculus called hyper tableaux in [1]. In this paper we propose a new calculus called hyperS tableaux which overcomes substantial drawbacks of hyper tableaux. Contrast to hyper tableaux, hyperS tableaux are entirely automated and heuristic. We prove the soundness and the completeness of hyperS tableaux. HyperS tableaux are applied in the theorem prover Sofia, which additionally provides useful tools for clause set generation (based on justificational tableaux) and for tableau simplification (based on redundancy), and advantageous heuristics as well. An additional feature is the support of the so-called parametrized theorems, which makes the prover able to give compound answers.

1 Introduction

Several syntactic methods have been constructed and implemented to automate theorem proving in first-order logic. Most of these methods can be classified as either tableau-based or resolution-based. Semantic tableaux were introduced by [8] and meet the problem of term selection for instantiating γ-formulas, which was tried to be overridden by the application of the most general unifier atomic closure rule for free-variable tableaux in [2]. Resolution acts with clauses. An easy-to-implement variant is SLD-resolution on which the programming language Prolog is based. SLD-resolution restricted to the class of Horn clauses is complete. One improved variant is the positive (negative) hyper-resolution which is not restricted and resolves the entire clause head (body) in a single inference step [7]. Hyper tableaux by [1] combine the advantageous features of positive hyper-resolution and clause tableaux [3], but admit some unwanted solutions which were tried to be eliminated in [4].

In this paper, we propose an improved and heuristic variant of rigid hyper tableaux calculus defined in [4], what we call *hyperS tableaux*. The name comes from the theorem prover *Sofia* in which the proposed calculus is applied and refers to that more than one clauses can be instantiated in a single inference step. After

*Department of Computer Science, University of Debrecen, Hungary. Email: kovasz@inf.unideb.hu

introducing the necessary concepts in Section 2, we will give details of hyperS tableaux in Section 3, where the soundness and completeness of the calculus will be proven. In Section 4.1, an easy-to-use and effective tableau-based method is introduced for generating clauses. In Section 4.2, the heuristical management of hyperS tableaux is argued in order to reduce the size of the tableau constructed. In Section 5, this issue will be further analyzed by recommending some simplifications in the tableau. These simplifications are related to the concept of redundancy [1]. Last, we mention the support for parametrized theorems, which is considered in Section 4.3.

2 Preliminaries

In the followings, we assume the reader to be familiar with the basic concepts of first-order logic.

Definition 1 (Equivalence). Two formulas A and B are *logically equivalent*, denoted by $A \equiv B$, iff in any model A is true iff B is true.

Definition 2 (Literal). A formula L is a *literal* iff $L = A$ or $L = \neg A$ where A is atomic.

Definition 3 (Positive/negative literal). Let L be a literal.

(1) If L is atomic and $L \notin \{\top, \bot\}$ then L is positive.

(2) If $L = \neg A$ where A is atomic, L is positive iff A is negative.

We have given an inductive definition of positive/negative literals. The definition is incomplete since we have not defined if $\top$ and $\bot$ are positive or negative. This must also be defined for a complete definition, and can be defined on demand, as it will be in Section 3 in connection with positive and negative hyper tableaux.

Definition 4 (Complement). A literal $\overline{L}$ is a *complement* of a literal L if

- $\overline{L} = \top$ if $L = \bot$, or
- $\overline{L} = \bot$ if $L = \top$, or
- $\overline{L} = \neg A$ if $L = A$, or
- $\overline{L} = A$ if $L = \neg A$,

where A is atomic.

Definition 5 (Clause). A *clause* is a set $C = \{L_1, \ldots, L_n\}$ where $n \geq 1$ and L_i $(1 \leq i \leq n)$ is a *literal*. C can be regarded as a disjunction of its literals, namely $L_1 \vee \cdots \vee L_n$. Let $\{\bot\}$ be the *empty clause*, and be denoted by $\square$.

In the literature, a tableau is usually defined as a labelled tree. However, a tableau is commonly regarded, and used, as a set of its branches, which are defined as sets of their formulas.

Definition 6 (Branch). A *branch* is a multiset $\mathcal{B} = \{L_1, \ldots, L_n\}$ where $n \geq 1$ and L_i $(1 \leq i \leq n)$ is a literal. $\mathcal{B}$ can be regarded as a conjunction of its literals, namely $L_1 \wedge \cdots \wedge L_n$.

Definition 7 (Tableau). A *tableau* is a multiset $\mathcal{T} = \{\mathcal{B}_1, \ldots, \mathcal{B}_n\}$ where $n \geq 1$ and $\mathcal{B}_i$ $(1 \leq i \leq n)$ is a branch.

The instantiation of a clause C in a tableau means that for some substitution σ (the literals of) $C\sigma$ is being attached to a branch of the tableau. We require to classify each parameter of a clause either as a *parametric variable* or as a *universal variable* (see Section 4.3).

Definition 8 (Brand new clause instance). A *new instance* of a clause C is $C\sigma$ where σ is a renaming substitution such that $Dom(\sigma)$ contains all universal variables in C, as defined in [3]. By a *brand new instance* of C, we mean a new instance $C\sigma$ where $Range(\sigma)$ consists of only "brand new" variables, i.e., variables that have not occured in the given derivation yet (neither in the clause set being refuted nor in the tableau being constructed). We say that a variable is *rigid* iff it was introduced by a brand new clause instance.

Definition 9 (Renamed variants). Two formula F_1 and F_2 are *renamed variants* iff there is a renaming substitution σ such that $F_1 = F_2\sigma$.

The concepts "unifiable" and "unifier" are well-known. We define the following concept:

Definition 10 (Complementary unifier). Two literals L_1 and L_2 are *complementary unifiable* iff there is a substitution σ such that $L_1\sigma$ is a complement of $L_2\sigma$, and σ is called a *complementary unifier* of L_1 and L_2.

3 HyperS Tableaux

Let $\mathcal{C}$ be a clause set. We assume that no $C \in \mathcal{C}$ contains any parametric variable.[1] The effects and advantage of eliminating this restriction will be explored in Section 4.3.

Definition 11 (Extension). An *extension* is a tuple (E, θ) where E is a clause and θ is a substitution.

Definition 12 (Extension application). Applying extension (E, θ) to a branch $\mathcal{B}$ in a tableau $\mathcal{T}$ means executing the following steps:

(1) $\mathcal{T} := \mathcal{T} \backslash \mathcal{B} \cup \{\mathcal{B} \cup \{L\} \mid L \in E\}$;

(2) $\mathcal{T} := \mathcal{T}\theta$.

[1] By this we refer to closed clauses, as it is known in the literature.

The following definitions (Definition 13, Definition 14, and Definition 15) and theorem (Theorem 19) concern *positive hyperS tableaux*. For negative hyperS tableaux, we can easily formulate the dual forms of these definitions and theorem only by switching all the adjectives "positive" to "negative", and vice versa. This is why we postponed the completion of Definition 3 to this section, which is done by the following definition.

Definition 13. Let $\top$ be negative, and $\bot$ be positive.

The following two definitions are for computing the possible extensions for a branch and a clause set.

Definition 14. Let C be a clause and let $\{L_1^{\ominus}, \ldots, L_n^{\ominus}\}$ $(n \geq 0)$ be the set of all negative literals in C. Furthermore, let $\{C_1, \ldots, C_n\}$ be a clause multiset and let $\{L_1^{\oplus}, \ldots, L_n^{\oplus}\}$ be a multiset of positive literals such that $L_i^{\oplus} \in C_i$ $(1 \leq i \leq n)$. The extension $\mathcal{E}$ for C, $\{C_1, \ldots, C_n\}$, and $\{L_1^{\oplus}, \ldots, L_n^{\oplus}\}$ is computed by executing the following steps:

(1) $E := \square$ and $\theta := \epsilon$;

(2) for all i $(1 \leq i \leq n)$:

  (a) let σ be the most general complementary unifier of $L_i^{\ominus}\theta$ and $L_i^{\oplus}\theta$;

  (b) if σ does not exist then $\mathcal{E}$ does not exist (halt);

  (c) $E := E \cup C_i'$ where $C_i' = C_i \backslash \{L_i^{\oplus}\}$;

  (d) $\theta := \theta\sigma$;

(3) $E := E \cup C'$ where $C' = C \backslash \{L_1^{\ominus}, \ldots, L_n^{\ominus}\}$;

(4) $\mathcal{E}$ exists and is (E, θ).

Definition 15 (Extension for branch and clause set). An extension for a branch $\mathcal{B}$ and the clause set $\mathcal{C}$ is an extension for

(1) any $\underline{C}$ where $C \in \mathcal{B}' \cup \mathcal{C}$ and $\mathcal{B}'$ is the set of the negative literals in $\mathcal{B}$,

(2) any multiset $\{\underline{C}_1, \ldots, \underline{C}_n\}$ where n is the number of negative literals in C and $C_i \in \mathcal{B} \cup \mathcal{C}$ $(1 \leq i \leq n)$, and

(3) any multiset $\{L_1, \ldots, L_n\}$ where L_i is a positive literal in $\underline{C}_i$ $(1 \leq i \leq n)$,

where for any clause D

$$\underline{D} = \left\{ \begin{array}{ll} \text{a brand new instance of } D & \text{, if } D \in \mathcal{C} \\ D & \text{, otherwise} \end{array} \right\}$$

Definition 16 (HyperS tableaux). For a clause set $\mathcal{C}$, a *hyperS tableau* is constructed as follows:

(1) (Initialization rule) A tableau consisting of a single branch $\{\top\}$ is a hyperS tableau for $\mathcal{C}$.

(2) (Extension rule) Let $\mathcal{T}$ be a hyperS tableau for $\mathcal{C}$, let the branch $\mathcal{B} \in \mathcal{T}$, and let $\mathcal{E}$ be an extension for $\mathcal{B}$ and $\mathcal{C}$. The tableau constructed by the application of $\mathcal{E}$ to $\mathcal{B}$ in $\mathcal{T}$ is a hyperS tableau for $\mathcal{C}$.

For proving hyperS tableaux to be sound and complete, we will use the soundness and completeness of *clause tableaux* introduced in [3].

Definition 17 (Clause tableaux). For the clause set $\mathcal{C}$, a clause tableau is constructed with the following rules:

(1) (Initialization rule) A tableau consisting of a single branch $\{\top\}$ is a clause tableau for $\mathcal{C}$.

(2) (Extension rule) Let $\mathcal{T}$ be a clause tableau for $\mathcal{C}$, let $\mathcal{B} \in \mathcal{T}$, and let $C \in \mathcal{C}$. Let $\{\underline{L}_1, \ldots, \underline{L}_n\}$ be a brand new instance of C. $\mathcal{T} \backslash \mathcal{B} \cup \{\mathcal{B} \cup \{\underline{L}_i\} \mid 1 \leq i \leq n\}$ is a clause tableau for $\mathcal{C}$.

(3) (Closure rule) Let $\mathcal{T}$ be a clause tableau for $\mathcal{C}$, let $\mathcal{B} \in \mathcal{T}$, and let the literals $L_1, L_2 \in \mathcal{B}$. If L_1 and L_2 are complementarily unifiable with the most general unifier σ, then $\mathcal{T}\sigma$ is a clause tableau for $\mathcal{C}$.

Lemma 18. *Clause tableaux are sound and complete as proven in [3].*

Theorem 19 (Soundness and completeness). *HyperS tableaux are sound and complete.*

Proof. It is sufficient to prove that the application of each rule of clause tableaux can be simulated by the application of the rules of hyperS tableaux, and vice versa. In the proof, we do not distinguish two tableaux if they contain some closed branches additionally as compared to each other. Such a distinction is superfluous in terms of derivation.

The initialization rule of the two calculi is the same. For the rest:

(I) For clause tableaux (notation is from Definition 17):

(1) (Extension rule) Let C' denote $C \cup \{\bot\}$. Notice that $\top \in \mathcal{B}$ and $C' \equiv C$. Consider the extension for $\{\top\}$, $\{\underline{C}'\}$, and $\{\bot\}$, namely $(\underline{C}', \epsilon)$ where $\underline{C}'$ is a brand new instance of C'.

(2) (Closure rule) Assume that L_1 is negative. Consider the extension for $\{L_1\}$, $\{\{L_2\}\}$, and $\{L_2\}$, which is actually $(\square, \sigma)$.

(II) For hyperS tableaux, the proof for the extension rule follows (notation is from Definition 14 and Definition 16).

Apply the extension rule of clause tableaux for $\mathcal{T}$ and for

(1) $\mathcal{B}$ and C_1;

(2) $\mathcal{B} \cup \{L_1^{\oplus}\}$ and C_2;

(3) $\mathcal{B} \cup \{L_1^{\oplus}, L_2^{\oplus}\}$ and C_3;

$\vdots$

(n) $\mathcal{B} \cup \{L_1^{\oplus}, \ldots, L_{n-1}^{\oplus}\}$ and C_n;

(n+1) $\mathcal{B} \cup \{L_1^{\oplus}, \ldots, L_n^{\oplus}\}$ and C

one after the other.

Then for each i $(1 \leq i \leq n)$, consider the branch containing $L_i^{\oplus}$ and $L_i^{\ominus}$, to which the closure rule is applied, i.e., σ_i is applied to the tableau where σ_i is the most general complementary unifier of $L_i^{\oplus}$ and $L_i^{\ominus}$. Notice that $\theta = \sigma_1 \sigma_2 \ldots \sigma_n$.

□

As usual, a branch $\mathcal{B}$ is said to be closed if $\mathcal{B}$ contains complementary literals. In hyperS tableaux, it would be sufficient to define the *closeness* of $\mathcal{B}$ as follows: $\mathcal{B}$ contains both $\top$ and $\bot$. What is more, since a branch always contains $\top$, it would be completely sufficient to monitor whether $\bot$ has occurred in $\mathcal{B}$. This latter method is very similar to that is used in resolution, namely monitoring whether $\square$ has occurred.

Compared to hyper tableaux, hyperS tableaux do not require "*purifying substitutions*", which moreover were generated by guessing in [1]. This is handled by using rigid variables, similarly to [4]. However, we have made improvements to the method written there in order to avoid other unwanted solutions like "clause copies" and "factoring". Three evident improvements, as compared to [4], have been made as follows.

In Definition 14, $\{C_1, \ldots, C_n\}$ is a multiset, i.e., a clause may occur more than once. This means that a clause can be instantiated more than once during a single extension application. This is why "*clause copies*" are not needed in hyperS tableaux. For example, consider the clause set consisting of $E_1 = \{P(x), Q(x)\}$ and $E_2 = \{\neg P(a()), \neg P(f(x)), R(y)\}$. E_1 should be instantiated twice in order to instantiate E_2 (see [4] for details). In hyperS tableaux, an extension can be constructed for E_2, $\{E_1, E_1\}$, and $\{P(x), P(x)\}$ (we are passing over brand new instances this time).

In Definition 15, (1), C can be chosen from $\mathcal{B}' \cup C$, i.e., not only from the clause set, but also from the branch. This results in the elimination of "*factoring.*" For example, consider the clause set consisting of $E_1 = \{P(x), P(y)\}$ and $E_2 = \{\neg P(z)\}$. The tableau, after the instantiation of E_1, should be factored [5], otherwise an infinite tableau will be constructed. In hyperS tableaux, this is avoided by applying both the extension for E_2, $\{E_1\}$, and $\{P(x)\}$, and the extension for E_2, $\{P(y)\}$, and $\{P(y)\}$ (we are passing over brand new instances again). We want to emphasize that although we have eliminated factoring, it could be performed as a

simplification (i.e., tableau reduction) on demand. In Section 5, we will propose other simplifications in connection with redundancy.

In Definition 15, (2), C_i $(1 \leq i \leq n)$ can be from $\mathcal{B} \cup \mathcal{C}$, i.e., not only from the branch, but also from the clause set. Such a permissiveness was motivated by the observation that [1] and [4] took into consideration only the "past" (i.e., what literals have been attached to the branch), but the "future" (i.e., what literals may be attached to the branch further on) not at all. By such a *lookahead*, we intend to reduce the size of the constructed tableau (and so the execution time). This is one of the reasons why we use *heuristics* in ranking extensions for the branch and the clause set (see Section 4.2).

4 Theorem Prover Sofia

The theorem prover Sofia applies hyperS tableaux for theorem proving. Besides the use of the calculus, Sofia provides automated clause set generation (Section 4.1) and the heuristic management of hyperS tableaux (Section 4.2), and can give compound answers by using parametric variables (Section 4.3).

4.1 Clause Set Generation by Justificational Tableaux

In this section, we propose an easy-to-use and effective method for generating a clause set for a formula, i.e., a clause set which is satisfiable iff the given formula is satisfiable. The method is based on the so-called justificational tableaux. Justificational tableaux are similar to refutational tableaux except for the nature of the questions that can be answered. While refutational tableaux are for investigating whether a formula is unsatisfiable, justificational tableaux can answer whether a formula is valid. This difference manifests in the form of the tableau expansion rules. Justificational tableau expansion rules are the ones in Figure 1, where we use α, β, γ, and δ formulas as introduced in the unifying notation by [8].

$$\frac{\alpha}{\alpha_1 \mid \alpha_2} \qquad \frac{\beta}{\begin{array}{l}\beta_1\\ \beta_2\end{array}} \qquad \frac{\gamma}{\gamma(x)} \quad x \text{ is a new parameter} \qquad \frac{\delta}{\delta(f(x_1,\ldots,x_n))} \quad \begin{array}{l}f \text{ is a new function symbol}\\ \text{and } FV(\delta) = \{x_1,\ldots,x_n\}\end{array}$$

Figure 1: Justificational Tableau Expansion Rules

Every branch of a justificational tableau can be regarded as a disjunction of formulas, and a justificational tableau is a conjunction of its branches. The closeness of a branch means that the branch, as a disjunction, is valid, and the closeness of a tableau corresponds to the fact that the tableau, as a conjunction, is valid.

Justificational tableau expansion rules for α, β, and γ formulas are based on the same facts as refutational tableau expansion rules. Namely: in any model

(1) α is true iff both α_1 and α_2 are true;

(2) β is true iff any of β_1 and β_2 is true;

(3) γ is true iff $\gamma(x)$ is true.

The justification tableau expansion rule for δ formulas is based on the following fact: for a formula F, F is satisfiable iff F^{SK} is satisfiable where F^{SK} is a Skolem formula for F [9]. Our treatment of δ formulas results in performing skolemization "on the fly", i.e., there is no need to skolemize F before constructing a tableau for F.

We consider a tableau finished iff each of its formulas either is a literal or has already been used for applying a tableau expansion rule. Since we want to use justificational tableaux only for clause set generation, we do not allow reusing any formulas[2]. Consequently, a finished justificational tableau for a formula can be constructed in finitely many steps.

Note that if only literals are considered then a finished justificational tableau for a formula F is a clause set for F. Hence, it is reasonable to let Sofia operate with two tableaux: one finished justificational tableau $\mathcal{T}$ for F and one hyperS tableau for (the set of the branches of) $\mathcal{T}$. In Figure 2, a clause set $\mathcal{C}$ is generated for the formula set of problem No. 31 in [6], as an example.

4.2 Heuristics

In this section, we argue what heuristic is worth to use for ranking the extensions for a branch and a clause set. Let $\mathcal{E} = (E, \theta)$ be an extension for a branch $\mathcal{B}$ of a tableau $\mathcal{T}$ and for a clause set $\mathcal{C}$. The heuristic is reckoned as a total order over the set of extensions. When defining the heuristic, we primarily consider the number of new branches in $\mathcal{T}$ after applying $\mathcal{E}$ to $\mathcal{B}$, which is exactly $|E|$. We secondarily consider the number of rigid variables substituted by the application of $\mathcal{E}$.

Definition 20 (Heuristic). For two extensions (E_1, θ_1) and (E_2, θ_2), $(E_1, \theta_1) \leq (E_2, \theta_2)$ iff

(1) $|E_1| < |E_2|$ or

(2) $|E_1| = |E_2|$ and $|R_1| \leq |R_2|$
where $R_i = \{x | x \in Dom(\theta_i) \cap FV(\mathcal{T})\}$, $i \in \{1, 2\}$.

Sofia applies the minimal extension to $\mathcal{B}$ in $\mathcal{T}$. Of course, the total order over the set of extensions can be defined differently.

In Figure 3, a derivation for the clause set $\mathcal{C} = \{C_1, \ldots, C_6\}$ in Figure 2 can be seen. The derivation consists of three hyperS tableaux shown on the left in Figure 3. On the right, the extensions generated for the single open branch of the given tableau and $\mathcal{C}$ are enumerated, ordered by the proposed heuristic. To the (open branch of the) first tableau $\mathcal{E}_1$ is applied. To the (open branch of the) second tableau $\mathcal{E}_3$ is applied, which only appends $\bot$ to the open branch, thus a closed tableau gets constructed. Hence, $\mathcal{C}$ is unsatisfiable.

[2] As it is known, reusing γ formulas must be allowed for a complete proof procedure [2].

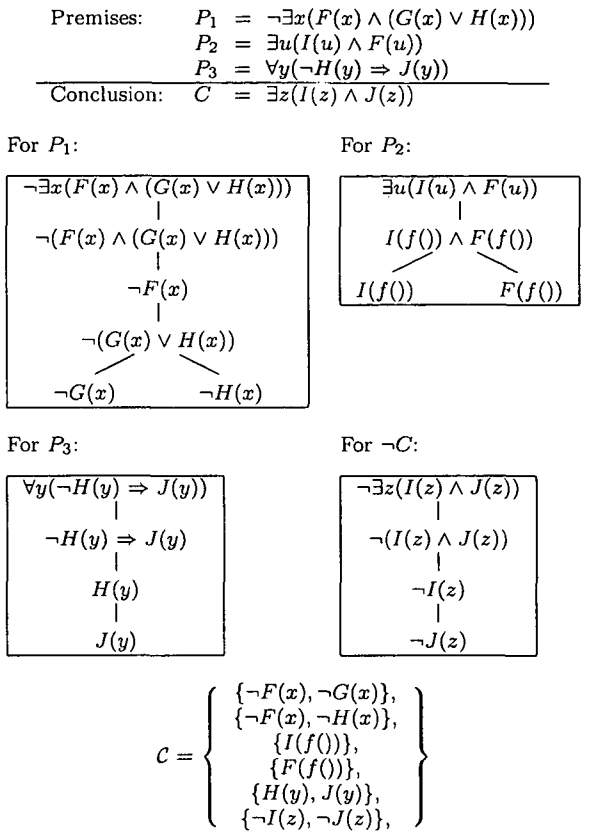


Figure 2: Generating a clause set.

4.3 Parametrized Theorems

As mentioned in Section 3, we remove the restriction on the closeness of clauses in a clause set $\mathcal{C}$ being refuted. Thus we let any clause in $\mathcal{C}$ contain parametric variables. For a user who wants the theorem prover to answer on the theoremhood of a formula F, the facility of permitting parametric variables in F provides exciting opportunities. The prover can give not only a yes/no answer but can tell the substitution θ to the parametric variables which makes $F\theta$ a theorem. Notice that F being a theorem is a special case of $F\theta$ being a theorem, when $\theta = \epsilon$.

As it is well-known, SLD-resolution (and hence, Prolog) allows the use of parametric variables. Actually, Prolog does not distinguish universal and parametric variables. The effect of this fact is the need of backtracking. The hyper tableaux calculus (and hence, the hyperS tableaux calculus) is a straightforward method (see [1]) since it prohibits the use of parametric variables. The explanation for this is that a substitution to (a rigid variable substituted to) a universal variable does

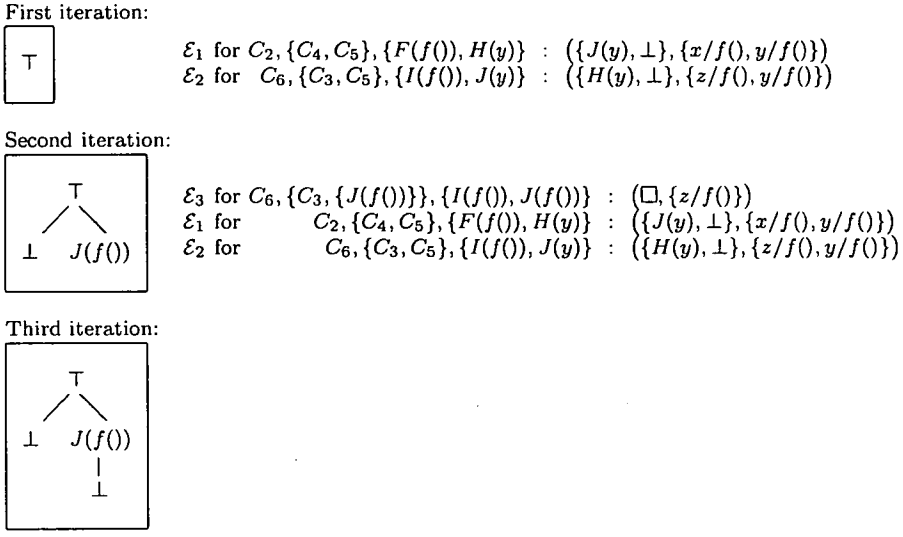


Figure 3: HyperS tableaux derivation.

not exclude the other substitutions to (a rigid variable substituted to) the same universal variable to be applied later, since new instances of the clauses in C can be attached to the tableau several times. Contrarily, a substitution to a parametric variable is final.

On the basis of the preceding discussion, the use of parametric variables in hyperS tableaux requires "rollback points" to be declared in a derivation whenever a parametric variable gets substituted, and to try to construct another derivation starting from the latter "rollback point" whenever a derivation ends in an open tableau. Thus, Sofia becomes a compromised solution: it substitutes universal variables in a straightforward way and it supports the use of parametric variables by backtracking.

5 Tableau Simplifications Based on Redundancy

In this section, we propose simplifications (or reductions) of the tableau being constructed, based on redundancy. [1] introduced the redundancy of a clause in a branch. We define the concept of redundancy by the following definition:

Definition 21 (Redundancy). A formula A is redundant in a formula B w.r.t. a logical connective $\circ$ iff $A \circ B \equiv B$.

In Section 5.1, we define the redundancy of an extension in a branch, based on [1] and [4], and then a simplification called the Redundancy Check in order to avoid repetitions along a branch.

In Section 5.2, we propose two other simplifications based on the redundancy of clauses.

5.1 Redundancy Check

Definition 22 (Redundancy in conjunction). A literal L is redundant in a conjunction of the literals $L_1, \ldots, L_n$ iff for some i $(1 \leq i \leq n)$ there is a substitution σ such that $L = L_i\sigma$.

Theorem 23. *If a literal L is redundant in a conjunction of the literals $L_1, \ldots, L_n$ then L is redundant in $L_1 \wedge \cdots \wedge L_n$ w.r.t. $\wedge$.*

Proof. By Definition 22, L is redundant in a conjunction of the literals $L_1, \ldots, L_n$ iff for some i $(1 \leq i \leq n)$ there is a substitution σ such that $L = L_i\sigma$.

By Definition 21, L is redundant in $L_1 \wedge \cdots \wedge L_n$ w.r.t. $\wedge$ iff $L_1 \wedge \cdots \wedge L_n$ and $L_1 \wedge \cdots \wedge L_n \wedge L$ are logically equivalent, i.e., in any model $L_1 \wedge \cdots \wedge L_n$ is true iff $L_1 \wedge \cdots \wedge L_n \wedge L$ is true. This latter equivalence must be proven.

Right-to-left is evident. For the reverse: in a model, $L_1 \wedge \cdots \wedge L_n$ is true iff L_j is true for all j $(1 \leq j \leq n)$. Hence, L_i is true. Since $L = L_i\sigma$, L is true. Hence, $L_1 \wedge \cdots \wedge L_n \wedge L$ is true. □

Definition 24 (Redundancy of clause in branch). A clause $\{L_1, \ldots, L_n\}$ is redundant in a branch $\mathcal{B}$ iff for some i $(1 \leq i \leq n)$ L_i is redundant in $\mathcal{B}$ as a conjunction.

Definition 25 (Redundancy of extension). An extension (E, θ) is redundant in a branch $\mathcal{B}$ iff $E\theta$ is redundant in $\mathcal{B}\theta$.

By using the redundancy of an extension, a simplification called the *Redundancy Check* can be introduced: do not apply an extension $\mathcal{E}$ to a branch B if $\mathcal{E}$ is redundant in $\mathcal{B}$.

5.2 Clause Simplifications

Definition 26 (Redundancy in disjunction). A literal L is redundant in a disjunction of the literals $L_1, \ldots, L_n$ iff for some i $(1 \leq i \leq n)$ there is a substitution σ such that $L\sigma = L_i$.

Theorem 27. *If a literal L is redundant in a disjunction of the literals $L_1, \ldots, L_n$ then L is redundant in $L_1 \vee \cdots \vee L_n$ w.r.t. $\vee$.*

Proof. By Definition 26, L is redundant in a disjunction of the literals $L_1, \ldots, L_n$ iff for some i $(1 \leq i \leq n)$ there is a substitution σ such that $L\sigma = L_i$.

By Definition 21, L is redundant in $L_1 \vee \cdots \vee L_n$ w.r.t. $\vee$ iff $L_1 \vee \cdots \vee L_n$ and $L_1 \vee \cdots \vee L_n \vee L$ are logically equivalent, i.e., in any model $L_1 \vee \cdots \vee L_n$ is false iff $L_1 \vee \cdots \vee L_n \vee L$ is false. This latter equivalence must be proven.

Right-to-left is evident. For the reverse: in a model, $L_1 \vee \cdots \vee L_n$ is false iff L_j is false for all j $(1 \leq j \leq n)$. Hence, L_i is false. Since $L_i = L\sigma$, L is false. Hence, $L_1 \vee \cdots \vee L_n \vee L$ is false. □

Definition 28 (Redundant clause). A clause C is redundant iff L is redundant in $C \backslash L$ as a disjunction for some $L \in C$.

The next definition and theorem are for simplifying a redundant clause to a non-redundant one.

Definition 29 (Non-redundant variant). A clause C' is a non-redundant variant of a clause C iff $C' \subseteq C$, $C' \equiv C$, and C' is not redundant.

Theorem 30.

(I) Of any clause, a non-redundant variant exists.

(II) If there are more than one such variants then they are renamed variants of each other.

Proof. Let C be a clause $\{L_1, \ldots, L_n\}$. Let us define the following function on clauses:

$$r(D) = \big|\{L \mid L \in D \text{ and } L \text{ is redundant in } D \backslash L \text{ as a disjunction}\}\big|.$$

(I) The proof is inductive on $r(C)$.

(1) If $r(C) = 0$ then C itself is a non-redundant variant of C since $C \subseteq C$, $C \equiv C$, and C is not redundant.

(2) Assume that $r(C) \geq 1$, and there is a non-redundant variant of any clause D if $r(D) < r(C)$. Prove that there is a non-redundant variant of C.

By Definition 28, there is an $L \in C$ such that L is redundant in $C' = C \backslash L$ as a disjunction. Since $r(C') < r(C)$, there is a non-redundant variant C'' of C', i.e., $C'' \subseteq C'$, $C'' \equiv C'$, and C'' is not redundant. By Theorem 27 and Definition 21, $C' \equiv C$. Since $C'' \subseteq C' \subseteq C$, $C'' \equiv C' \equiv C$, and C'' is not redundant, C'' is a non-redundant variant of C.

(II) In the proof for (I), an inductive method for computing non-redundant variants is used, where literals are eliminated in C one by one until a non-redundant clause gets computed. Let us prove that it does not matter in what order they are eliminated, i.e., the possible resulting clauses are renamed variants.

It is sufficient to prove that the elimination of two literals results either in one single clause or in renamed variants. That is, the case when $r(C) \geq 2$ is focused. Without loss of generality, it can be assumed that L_1 is redundant in the disjunction of $\{L_2, \ldots, L_n\}$, and L_2 is redundant in the disjunction of $\{L_1, L_3, \ldots, L_n\}$. By Definition 26, $L_1\sigma = L_i$ for some $i \in \{2, \ldots, n\}$ and some σ, and $L_2\theta = L_j$ for some $j \in \{1, 3, \ldots, n\}$ and some θ. There are three distinct cases:

(1) $i, j \notin \{1, 2\}$: both L_1 and L_2 are eliminated, thus the resulting clause is $\{L_3, \ldots, L_n\}$.

(2) $i \notin \{1,2\}$ and $j = 1$: $L_2\theta = L_1$, hence $L_2\theta\sigma = L_i$, which leads to the previous case.

(3) $i = 2$ and $j = 1$: $L_1\sigma = L_2$ and $L_2\theta = L_1$, hence $L_1\sigma\theta = L_1$. This holds iff $\sigma\theta = \epsilon$, which holds iff σ and θ are renaming substitutions. Thus the possible resulting clauses $\{L_2, L_3, \ldots, L_n\}$ and $\{L_1, L_3, \ldots, L_n\}$ are renamed variants.

□

Based on Theorem 30, we propose two simplifications:

(1) As an initializing step, simplify all the clauses in the clause set C you want to refute, i.e., try to refute the clause set that consists of non-redundant variants, exactly one of each clause in C.

(2) As a simplification of an extension (E, θ), replace E with one of its non-redundant variants.

6 Conclusion

In this paper, we proposed a new calculus called hyperS tableaux, which is a refinement of the hyper tableaux calculus. HyperS tableaux eliminate the non-automatable and unwanted solutions in hyper tableaux, automate the clause set generation, and perform a kind of heuristic lookahead. We proved the soundness and completeness of this calculus. We proposed several facilities based on redundancy for tableau reduction. Furthermore, we discussed the requirements for using parametric variables, which are necessary for a theorem prover to be able to give compound answers.

References

[1] P. Baumgartner, U. Furbach, I. Niemelä, "Hyper Tableaux". *Proc. JELIA '96*, Vol. 1226 of LNAI, Springer, 1996.

[2] M. Fitting, "First-Order Logic and Automated Theorem Proving". Springer-Verlag, 1996.

[3] R. Hähnle, "Tableaux and Related Methods", *Handbook of Automated Reasoning*, by J. A. Robinson and A. Voronkov, Vol. 1, Chapter 3, p. 100-178, Elsevier and MIT Press, 2001.

[4] M. Kühn, "Rigid Hypertableaux", *Proc. KI '97: Advances in Artificial Intelligence*, Vol. 1303 of LNAI, Springer, 1997.

[5] R. Letz, K. Mayr, C. Goller, "Controlled Intergration of the Cut Rule into Connection Tableau Calculi", *Journal of Automated Reasoning*, Vol. 13, p. 297-328, 1994.

[6] F. J. Pelletier, "Seventy-Five Problems for Testing Automatic Theorem Provers", *Journal of Automated Reasoning.* Vol. 2, p. 191-216, 1986.

[7] J. A. Rosinson, "Automated Deduction with Hyper-Resolution", *International Journal of Computer Mathematics.* Vol. 1, p. 227-234, 1965.

[8] R. M. Smullyan, "First-Order Logic". Springer-Verlag, 1968.

[9] K. Pásztorné Varga, M. Várterész, "A matematikai logika alkalmazásszemléletû tárgyalása". Panem, 2003.

Acta Cybernetica **17** (2005) 339–357.

Constraint Validation Support in Visual Model Transformation Systems

László Lengyel*, Tihamér Levendovszky*, and Hassan Charaf*

Abstract

Model-Driven Architecture (MDA) standardized by OMG facilitates to separate the platform independent part and the platform specific part of a system model. Due to this separation Platform-Independent Model (PIM) can be reused across several implementation platforms of the system. Platform-Specific Model (PSM) is ideally generated automatically from PIM via model transformation steps. Because of the appearance of high level languages, object-oriented technologies and CASE tools, metamodeling becomes more and more important. Metamodeling is one of the most central techniques both in design of visual languages, and reuse existing domains by extending the metamodel level. The creation of model compliers on a metamodeling basis is illustrated by a software package called Visual Modeling and Transformation System (VMTS), which is an n-layer multipurpose modeling and metamodel-based transformation system. VMTS is able to realize an MDA model compiler. This paper (i) addresses the relationship between the constraints enlisted in metamodel-based rewriting rules and the pre- and postconditions, (ii) it introduces the concepts of general validation, general preservation and general guarantee, which facilitate that if a transformation step is specified adequately with the help of constraints, and the step has been executed successfully for the input model, then the generated output model is in accordance with the expected result, which is described by the transformation step refined with the constraints.

An illustrative case study based on constraint specification in rewriting rules is also provided.

1 Introduction

OMG's Model Driven Architecture [17] offers a standardized framework to separate the essential, platform independent information from the platform dependent constructs and assumptions. A complete MDA application consists of a definitive platform-independent model (PIM), and one or more platform-specific models (PSM) and complete implementations, one on each platform that the application

*Budapest University of Technology and Economics, 1111 Budapest, Goldmann György tér 3., Hungary, e-mails: lengyel@aut.bme.hu, tihamer@aut.bme.hu, hassan@aut.bme.hu

developer decides to support. The platform independent artifacts are mainly UML and other software models containing enough specification to generate the platform dependent artifacts automatically by so-called model compilers. Hence software model transformation provides a basis for model compilers, which plays central role in the MDA architecture.

Model transformation means converting an input model that is available at the beginning of the transformation process to an output model. MDA sets out a more restrictive definition: the output model should describe the same system as the input model. But our approach (VMTS [12] [25]) has been designed to be able to specify more general transformations. Model compilers can support properties to guarantee, preserve or validate them, and the presented approach is a practical application of these mechanisms. Models can be considered special graphs; simply contain nodes and edges between them. This mathematical background makes possible to treat models as labeled graphs and to apply graph transformation algorithms to models. The steps of graph transformation are rewriting rules, each rewriting rule consists of a left-hand side graph (LHS) and right-hand side graph (RHS). Previous work [13] has introduced an approach, where LHS and RHS of the rules are built from metamodel elements. It means that an instantiation of the LHS must be found in the graph to which the rule is applied (host graph) instead of the isomorphic subgraph of the LHS. Hence the LHS and RHS graphs are the metamodels of the graphs which we search and replace in the host graph.

Often it is not enough to match graphs based on the topological information only, we want to restrict the desired match by other properties, e.g. we want to match a subgraph with a node, which has a special property or which has a unique relation between the properties of the matched nodes. For example we want to match a state in a statechart model which has at least one incoming and two outgoing transitions. The metamodel-based specification of the rules [13] allows assigning OCL [18] constraints to the rules, using the guidelines of the UML standard [19]. Because these constraints are bound to the rules, they are able to express local constraints. This is inherently a local construct, because the elements not appearing in LHS or RHS cannot be directly included in the OCL statements. Although the specification has this local-nature, it does not mean that validating them does not involve checking other model elements in the input model: constraint propagation needs to be taken into account by both the algorithmic background and the user of the transformation on specifying the constraint. OCL constraints which are enlisted in the LHS and RHS graphs affect the matched instances of the LHS and RHS graphs.

We have pre- and postconditions and OCL constraints assigned to the rewriting rules. In this paper we introduce the relation between them and the applicability of this concept based on a case study.

2 Backgrounds and Related work

The purpose of contracts [16] is to help us build better software by organizing the communication between software elements through specifying the mutual obligations. Contracts are used to guarantee that these communications occur on the basis of precise specifications of what these services are going to be. For the software to be able to guarantee any kind of correctness and robustness properties, they must know the precise constraints over such communications. In a client/supplier relationship, where the client needs a certain service and the supplier provides that service, to impose certain obligations on the client as to the kind of original program state that is permissible, when the client calls the supplier or the kind of arguments that the client routine passes to the supplier. These are preconditions, and they are obligations for the client. In the other direction, we are going to express the conditions that the supplier routine must guarantee to the client on completion of the supplier's task. That is the postcondition of the contract, specifically, the postcondition of that particular routine. The postcondition is also an obligation for the supplier. Besides the pre- and postcondition the third fundamental element of contracts is the invariant. A class invariant is a condition that applies to an entire class. It describes a consistency property that every instance of the class must satisfy.

The Object Constraint Language [18] is a formal language for analysis and design of software systems. It is a subset of the industry standard Unified Modeling Language [19] that allows software developers to write constraints and queries over object models. A constraint is a restriction on one or more values of an object-oriented model or system. There are four types of constraints: (i) An invariant is a constraint that states a condition that must always be met by all instances of the class, type, or interface. (ii) A precondition to an operation is a restriction that must be true at the moment that the operation is going to be executed. The obligations are specified by postconditions. (iii) A postcondition to an operation is a restriction that must be true at the moment that the operation has just ended its execution. (iv) A guard is a constraint that must be true before a state transition fires. Besides these, OCL can be used as a navigation language as well.

Graph rewriting [2, 7, 21] is a powerful tool for graph transformations with strong mathematical background. Originally it was developed as the natural generalization of Chomsky grammars to generate and parse visual languages. Instead of that graph language approach we will use the mechanism of the individual parsing steps, so-called rewriting rules for graph transformations. Rewriting rules are the atoms of graph transformation, which consists of a left hand side graph (LHS) and right hand side graph (RHS). Applying a graph rewriting rule means finding an isomorphic occurrence (match) of the LHS in the graph to witch the rule being applied (host graph), and replacing this subgraph with RHS. Replacing means removing elements which are in the LHS but not in the RHS, and gluing elements which are in the RHS but not in the LHS. Replacing process consists of two steps: removing and gluing, this approach is the so-called double pushout (DPO) [21]. The graph transformation is defined as an ordered sequence of rewriting rules, in other

words we control the transformation process by sequencing the rewriting rules. The rewriting rule is a stereotype of the UML activity state (<<Rewriting Rule>>) [13].

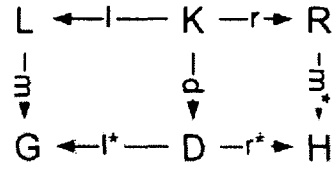


Figure 1: An illustration of direct derivation: L, K, R are the left-hand side, the interface and the right hand side graph; G and H are the graphs before and after the rule firing and D corresponds to K; l, r, l*, r*, m, d, m* are inclusions

The DPO approach accomplishes the rule firing by two steps: after finding a redex (the part of the host graph parsed by the rewriting rule), the first step removes the elements (vertices and edges) from the redex which are in the redex, but not in the RHS graph. This modified redex is referred to as interface graph. Then as a second step the elements of the RHS graph not in the interface graph but in the RHS graph are glued to the interface graph. The rewriting rule is characterized by a double pushout. The application of the rules results in a direct derivation of the host graph (Fig. 1). The category theory framework provides more flexible and more general background, so the DPO approach can be applied to many graph-like categories. For labeled and directed graphs the existence of the pushout (which is the condition to fire a rule) can be ensured by forcing the so-called gluing condition. The gluing condition consists of two parts. Firstly, the identification condition, which states that different vertices in the rewriting rule cannot match the same vertex in the host graph. Secondly, the dangling edge condition has to be dealt with as well: if a vertex should be deleted which is connected to an edge that is not inside the redex, the production rule cannot be fired. Unfortunately, this makes impossible to delete a connected vertex without considering its environment. Related to the DPO approach a rather tutorial like description can be found in [3, 5, 6, 4], and a more complete summary in [8, 21].

Our tool set, the Visual Modeling and Transformation System (VMTS) [13, 25] is an n-layer multipurpose modeling and metamodel-based transformation system. Using this environment, it is easy to edit metamodels, design models according to their metamodels, transform models using graph rewriting [13, 14]. It facilitates to check the metamodel constraints during the metamodel instantiation, and the rewriting rule constraints during the graph transformation process. VMTS can also be used to parse visual languages. The control structure can be thought of as a similar construct to programmed graph rewriting systems e.g. [20] and the rewriting process follows rules of the DPO direct derivation. Although the rewriting rules are specified in terms of the metamodel elements [13], on the instantiation level the rule firing mechanism is equivalent to the DPO rewriting rules.

VMTS benefits from the results of the mathematical background of formal lan-

guages, graph rewriting and results related to the metamodel-based software model transformation. It also incorporates several ideas from other existing environments [1, 10, 15], which implement the OCL, and enable constraints to be checked over models.

Model Integrated Computing (MIC) [22, 23, 24] is a model-based approach to software development, facilitating the synthesis of application programs from models created using customized, domain-specific program synthesis environments. MIC focuses on models, supports the flexible creation of modeling environments, and helps following the changes of the models. At the same time it facilitates code generation and provides tool support for turning the created models into code artifacts. Metamodeling environments and model interpreters together form the tool support for MIC. So far MIC is the only methodology, which requires metamodeling environments, model processors and provides a framework for them to cooperate to create Computer-Based Systems (CBS) in the practice. VMTS implements the ideas of MIC and it is able to realize an MDA model compiler.

The Generic Modeling Environment (GME) [9] is a metamodeling tool from which our system has borrowed several concepts of a metamodel-based modeling tool. GME on its own is not a transformation system, although the underlying MultiGraph Architecture (MGA) can be reached from the GReAT transformation system. GME supports constraint handling, it has a constraint interpreter called Constraint Manager.

The GReAT framework [11] is a transformation system for domain specific languages (DSL) built on metamodeling and graph rewriting concepts; it uses a proprietary notation and interpretation instead of instantiation between the rules expressed with meta elements and the match. The sequencing of the rewriting rules and parameter passing are similar in GReAT and in VMTS.

PROGRES [20] is a visual programming language in the sense that it has a graph-oriented data model and a graphical syntax for its most important language constructs. There are rather few tools which support pre- and postconditions, but VMTS and PROGRES manage them. In PROGRES the precondition of a transaction is a query, which should never fail, applied to the input graph of the surrounding transaction. Similarly the postcondition of a transaction is a query, which should never fail applied to the output graph of the surrounding transaction. It is allowed to access the in- and out-parameters of its transaction, but does not distinguish between a before- and an after-state of referenced nodes.

3 Contributions

The design by contract principle and model transformation by graph rewriting mechanism has been successfully applied in checking software systems. Thus revealing the connection between the concepts of pre- and postconditions and the OCL constraints assigned to the rewriting rules seems a promising direction.

This section (i) introduces the relation between the pre- and postconditions and OCL constraints assigned to the rewriting rules, (ii) presents how to check models

based on this relation, and (iii) discusses a metamodel-based approach along with a simple but illustrative case study: while we give and explain our definitions and propositions, we can show their benefit immediately on a practical example.

3.1 Relation between the Pre- and Postconditions and OCL Constraints

For the unified treatment we give the basic definitions, which are mainly based on [16]:

Definition 1. (*Transformation and Finite sequence of steps*)
A *Transformation* and a *Finite sequence of steps* consist of n number of rewriting rules (where $n > 0$) in an ordered sequence. This sequence defines the execution order of the contained rewriting rules. The difference between a *transformation* and a *finite sequence of steps* is that a *finite sequence of steps* is always comes to an end, it always terminates. A *transformation*, however, can contain infinite number of steps.

Definition 2. (*Precondition and Postcondition*)
A *precondition* (*postcondition*) assigned to a rewriting rule is a boolean expression that must be true at the moment when the rewriting rule is fired (after the completion of a rewriting rule). If a *precondition* of a rewriting rule is not true then the rewriting rule fails without being fired. If a *postcondition* of a rewriting rule is not true after the execution of the rewriting rule, the rewriting rule fails.

Definition 3. (*Validation, Preservation, Guarantee*)
Validation of a property: a transformation step S *validates* a property P specified by a boolean expression, when the following condition always holds: if a property P was true before the step S it remains true after the execution of the step S, and if P is false, the step S fails.
Preservation of a property: a transformation step S *preserves* a property P specified by a boolean expression, when the following condition always holds: if a property P was false (true) before the step S it remains false (true) after the execution of the step S.
Guarantee of a property: a transformation step S *guarantees* a property P specified by a boolean expression, when the following condition always holds: if a property P was true before the step S it remains true after the execution of the step S, and if P is false, the step S changes property P to true.

Definition 4. (*Pre value*)
If an OCL constraint is specified in the RHS of a transformation step, $x@pre$ means the value of x immediately before the rule was fired even if the value of x has not changed.

A direct corollary of Definition 2 that an OCL expression in LHS is a precondition to the rewriting rule, and an OCL expression in RHS is a postcondition to the

	property P before the step S	property P after the step S
Validation	true	true
	false	step S fails
Preservation	true	true
	false	false
Guarantee	true	true
	false	true

Table 1: Truth table of the validation, preservation and guarantee properties

rewriting rule. A rewriting rule can be fired if and only if all conditions enlisted in LHS are true. Also, if a rewriting rule finished successfully then all conditions enlisted in RHS must be true.

Table 1 illustrates the truth table of the validation, preservation and guarantee properties.

3.2 A Case Study

To show the applicability and the practical relevance of the results, a case study is provided. The case study contains a rewriting rule which uses a statechart model as input model and builds a CodeDOM [25] tree (an abstract syntax like graph representation of the code to be generated) in the VMTS database which we can use easily to generate source code from it on optional language (e.g. C++, C# or J#).

In Fig. 2 there is a statechart model of a water tank. Our case study uses this statechart model as input graph and applies a rewriting rule (Fig. 3) to it.

In VMTS it is possible that the LHS and the RHS of a rewriting rule have different metamodel. In the rewriting rule depicted in Fig. 3 the metamodel of the LHS is the Statechart metamodel [19, 25] and the metamodel of the RHS is the CodeDOM metamodel [25]. On the LHS of the rewriting rule there are two states, whose meta type is statechart state, and there is a transition between them with a 0..* multiplicity on the side of the target state. It means that exhaustively applying this rewriting rule to a statechart model, it will match all states with their target adjacent states. The rule has to match the accessible adjacent states because we need them to generate the state-transitions in the source code. Obviously it is possible that a state has no outgoing transitions, and the reason why we enable the 0 in the multiplicity is that we want to match states having only incoming transitions to generate CodeDOM tree for it. On the RHS of the rewriting rule the *CTypeDeclaration* represents a type declaration for a class, structure, interface or enumeration. *CMemberField* can be used to denote the declaration for a field of a type, and *CMemberMethod* to phrase the declaration for a method. *CParameter* represents a parameter declaration for a method, property, or constructor, and *CSnippetStatement* means a statement using a literal code fragment.

In a rewriting rule we can connect the LHS elements to the RHS elements, this

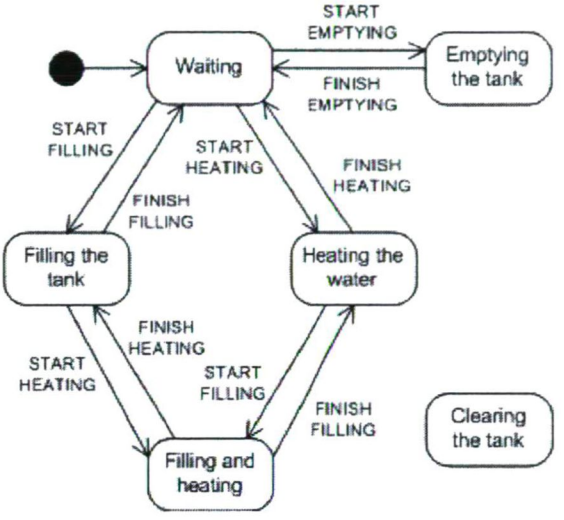


Figure 2: Case study: statechart of the water tank

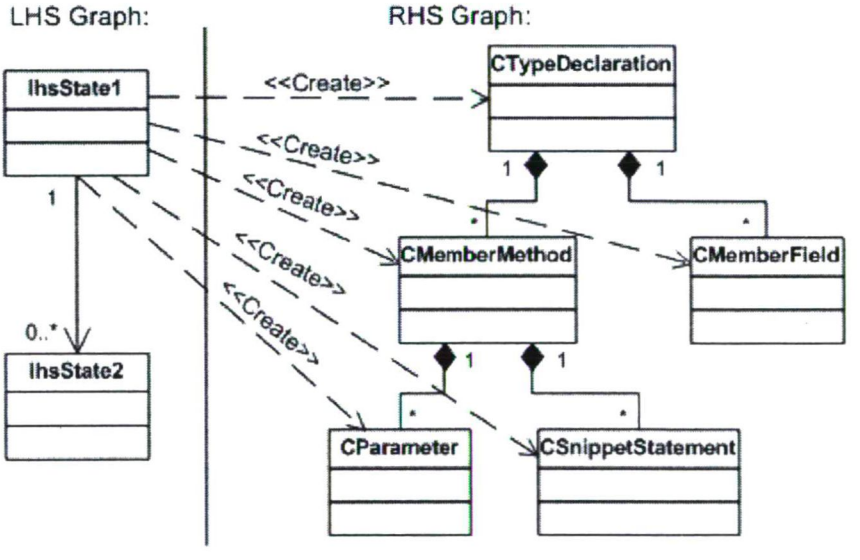


Figure 3: Rewriting rule of the case study

relation between LHS and RHS elements is called causality [11], which facilitates to assign an operation to this connection. Causalities can express either modification or removal of an LHS element, or creation of an RHS element. In Fig. 3 the causalities are denoted as dashed lines. The create operation and attribute transformation that are one of the most important part of the rewriting process are accomplished by XSL scripts. XSL scripts can access the attributes of the objects matched to LHS elements, and produce a set of attributes for the RHS element to which the causality point.

After the rewriting process the output graph has two parts. The difference between the two parts is that the first part is not affected, while the second part is affected by the rewriting rule. Accordingly we can examine the constraints contained by our rewriting rule only in the second part of the output graph, which is affected by the rewriting rule. Our case study is a fortunate case, because the whole right side and the whole output graph is generated. Hence it facilitates to check the local-nature constraints in the whole output graph.

In the next subsections we give our propositions for individual steps (rewriting rule), and for finite sequence of steps in along with the validation, preservation and guarantee properties, and we will illustrate their applicability based on the presented case study.

3.3 General Validation

Based on the validation property we can introduce the concepts of general validation. The goal of the general validation is that if a rewriting rule (a finite sequence of steps) is specified properly with the help of validation type constraints, and the rewriting rule (the finite sequence of steps) has been executed successfully for the input model, then the generated output model is in accordance with the expected result, which is described by the rewriting rule (the finite sequence of steps) refined with the constraints.

Proposition 1. General validation (Sufficient and necessary conditions)

(i) A step S validates a property P for an input model M if the property P is enlisted in both pre- and postconditions of the step S, and the step S has been executed successfully for the model M.

Proof. (S^{LHS} - is the left-hand side (LHS) of the step S, and S^{RHS}- is the right-hand side (RHS) of the step S). Assuming that the property P is enlisted in both S^{LHS} and S^{RHS}, and the step S has been executed successfully for the model M, but the step S does not validate the property P.

This is a contradiction because (1) if the property P is not true the step S cannot even be fired. (2) If the property P is true the step S can be fired. The property P is enlisted in S^{RHS}, and the step S has been executed successfully for the model M, it means that the property

P is true after the execution of the step S, which is equivalent to the definition of the validation. Hence the step S validates the property P.

(ii) If a step S validates a property P for a model M without conditions in the step S for the property P, then the property P can be enlisted in S^{LHS} and S^{RHS} without changing the result of the step S for the model M.

Proof. Assuming that (a) step S validates a property P for a model M without conditions in the step S for the property P, (b) one enlists the property P in S^{LHS} and S^{RHS}, then because of the newly added constraints, the result for the step S is different.

This is a contradiction because (1) if the property P is false: then the step S fails in both cases. In the first case (a) if the step S validates the property P and the property P is false, then the step S fails (definition of the validation). In the second case (b) if the property P placed in S^{LHS} and the property P is false, then the step S fails without being fired (definition of the precondition). (2) If the property P is true, the step S can be fired. In the first case (a) the property P remains true after the execution of the step S, because the step S validates the property P. In the second case (b) the property P is enlisted in S^{LHS} and S^{RHS}, which is equivalent to the definition of the validation. It means that the result can not be different.

(iii) A finite sequence of steps $(S_1, S_2 \ldots S_n)$ validates a property P for an input model M if the property P is enlisted both in preconditions of the step S_1 and in the postconditions of the step S_n, and the finite sequence of steps $S_1, S_2 \ldots S_n$ has been executed successfully for the model M.

(iv) If a finite sequence of steps $(S_1, S_2 \ldots S_n)$ validates a property P for an input model M without conditions in S_1^{LHS} and S_n^{RHS} for the property P, then the property P can be enlisted in S_1^{LHS} and S_n^{RHS} without changing the result of the finite sequence of steps $S_1, S_2 \ldots S_n$ for the model M.

As it is stated above, the rewriting rule of the case study generates a CodeDOM tree for the matched states. An example for the validation can be the following: the rewriting rule validates that the states with generated CodeDOM tree are not isolated (unreachable) states; it means that starting from the start state we can get at these states. To examine whether a state is unreachable, we enlist a constraint in the precondition of the rewriting rule, which checks whether the sate is isolated (Fig. 4). If the state is reachable (e.g. *Filling the tank*), the rewriting rule can be fired, but if the state is isolated (e.g. *Cleaning the tank*), the step fails (there is no proper match because of the violated constraint), as it is depicted in Fig. 5.

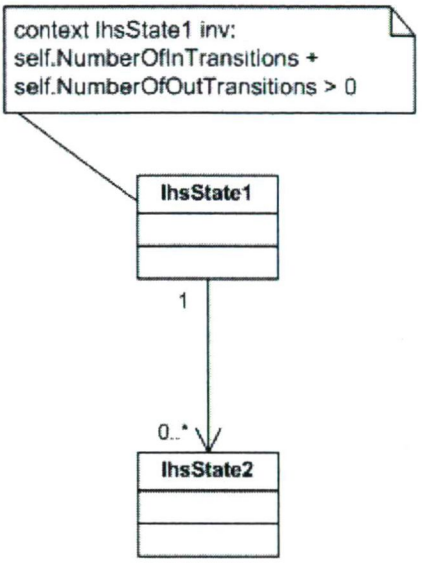


Figure 4: The LHS of the rewriting rule with an OCL constraint

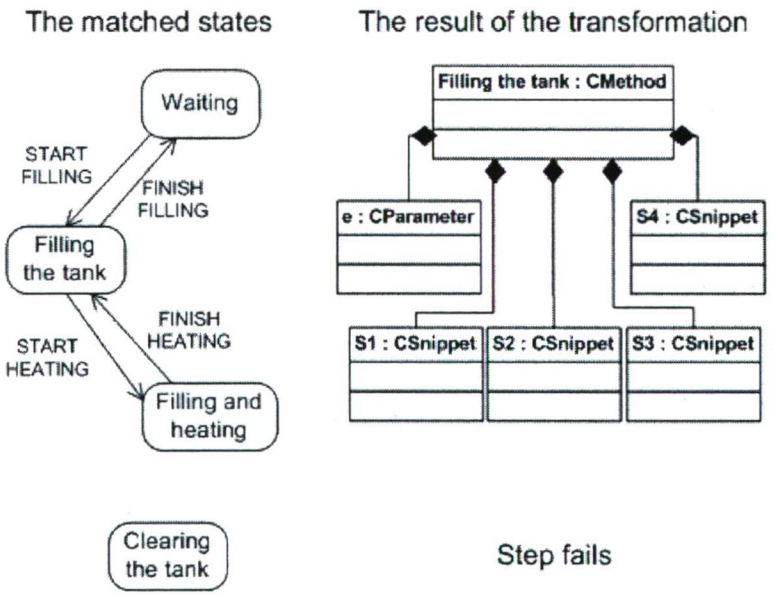


Figure 5: Examples for the matched states and the results of the transformation

3.4 General Preservation

Based on the preservation property we can introduce the concepts of general preservation. The modeler's task is to create adequate rewriting rules (finite sequence of steps) and specify them fully with constraints (pre- and postconditions). If the execution of the rewriting rule (the finite sequence of steps) finishes successfully, it preserves the required property values.

Proposition 2. General preservation (Sufficient and necessary condition)

(i) A step S preserves a property P for an input model M if the expression (*NOT P or P@pre*) *and* (*P or NOTP@pre*) is enlisted as an OCL expression in the postconditions of the step S, and the step S has been executed successfully for the model M.

Proof. Assuming that the expression (*NOT P or P@pre*) *and* (*P or NOT P@pre*) is enlisted as an OCL expression in S^{RHS}, and the step S has been executed successfully for the model M, but the step S does not preserve the property P.

This is a contradiction because the expression (*NOT P or P@pre*) *and* (*P or NOT P@pre*) is true if and only if the $P@pre = P$ holds after the execution of the step S, and it means that the step S preserves the property P.

(ii) If a step S preserves a property P for a model M without a postcondition in the step S for the property P, then the expression (*NOT P or P@pre*) *and* (*P or NOT P@pre*) as an OCL expression can be enlisted in S^{RHS}, and the step S also preserves the property P for the model M.

Proof. Assuming that (a) the step S preserves a property P for a model M without postconditions in the step S for the property P, (b) one enlists the expression (*NOT P or P@pre*) *and* (*P or NOT P@pre*) as an OCL expression in S^{RHS}, but in this case, because of the newly added constraint, the step S does not preserve the property P for the model M.

This is a contradiction, because in the first case (a) the step S preserves the property P. In the second case (b) the enlisted constraint is equivalent to the definition of the preservation, hence the step S preserves the property P in the second case (b) as well.

(iii) A finite sequence of steps (S_1, S_2 ... S_n) preserves a property P for an input model M if the expression (*NOT P or P@preS1*) *and* (*P or NOT P@preS*$_1$) is enlisted as an OCL expression in the postconditions of the step S_n, and the finite sequence of steps S_1, S_2 ... S_n has been executed successfully for the model M.

(iv) If a finite sequence of steps $(S_1, S_2 \ldots S_n)$ preserves a property P for an input model M without conditions in S_n^{RHS} for the property P, then the expression $(NOT\ P\ or\ P@preS_1)\ and\ (P\ or\ NOT\ P@preS_1)$ as an OCL expression can be enlisted in S_n^{RHS}, and the finite sequence of steps $S_1, S_2 \ldots S_n$ also preserves the property P for the model M.

The preservation property is important for the individual node attributes and for the relation between certain nodes. For example if a matched node has a property called *History* with the value *Deep*, then after rewriting the generated node which corresponds to the matched node also has a *History* property with the value *Deep* (Fig. 6).

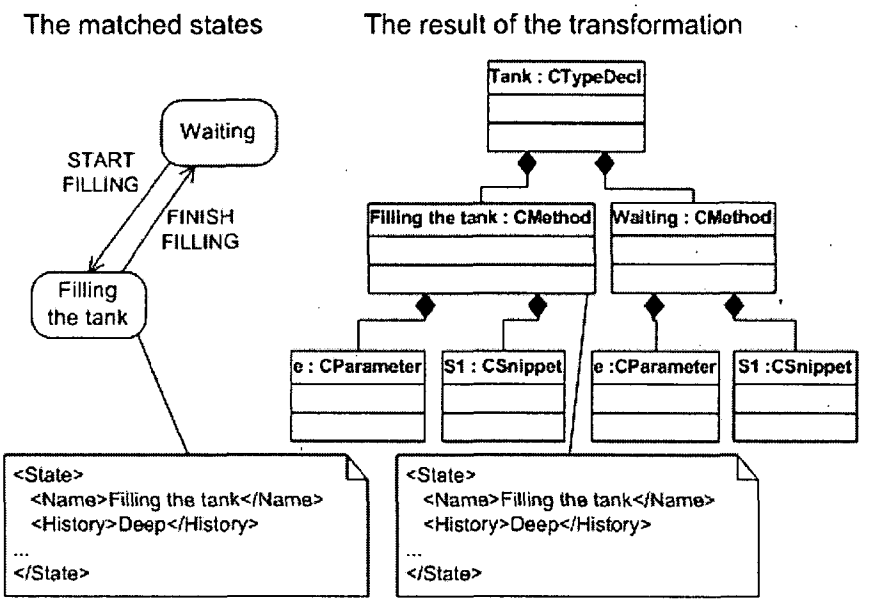


Figure 6: Example for the preservation of a property

3.5 General Guarantee

Similarly to the general validation and the general preservation, and using the guarantee property we can introduce the concepts of general guarantee.

Proposition 3. General guarantee (Sufficient and necessary condition)

(i) A step S guarantees a property P for an input model M if the property P is enlisted in the postconditions of the step S, and the step S has been executed successfully for the model M.

Proof. Assuming that the property P is enlisted in S^{RHS}, and the step S has been executed successfully for the model M, but the step S does not guarantee the property P.

This is a contradiction, because the property P is enlisted in S^{RHS}, and the step S has been executed successfully for the model M, and it means that the property P is true after the execution of the step S, which is equivalent to the definition of the guarantee. Hence the step S guarantees the property P.

(ii) If a step S guarantees a property P for a model M without a postcondition in the step S for the property P, then the property P can be enlisted in S^{RHS} and the step S also guarantees the property P for the model M.

Proof. Assuming that (a) step S guarantees a property P for a model M without postcondition in the step S for the property P, (b) one enlists the property P in S^{RHS}, but in this case, because of the newly added constraint, the step S does not guarantee the property P for the model M.

This is a contradiction, because in the first case (a) the step S guarantees the property P. In the second case (b) the enlisted property P is equivalent to the definition of guarantee, hence the step S also guarantees the property P in the second case (b).

(iii) A finite sequence of steps $(S_1, S_2 \ldots S_n)$ guarantees a property P for an input model M if (a) the property P is enlisted in the preconditions of step S_i (where $1 \leq i \leq n$), and $S_i, S_{i+1} \ldots S_n$ preserve the property P, or (b) the property P is enlisted in the postconditions of step S_i (where $1 \leq i \leq n$), moreover $S_{i+1}, S_{i+2} \ldots S_n$ preserve the property P, and the finite sequence of steps $S_1, S_2 \ldots S_n$ has been executed successfully for the model M.

(iv) If a finite sequence of steps $(S_1, S_2 \ldots S_n)$ guarantees a property P for an input model M without conditions in S_n^{RHS} for the property P, then the property P can be enlisted in S_n^{RHS}, and the finite sequence of steps $S_1, S_2 \ldots S_n$ also guarantees the property P for the model M.

In the statechart model of the case study every state has a property called *IsCodeAlreadyGenerated*. If this property is true, it means that this state already has a generated CodeDOM model. Exhaustively executing our rewriting rule, it guarantees that the *IsCodeAlreadyGenerated* property of every state - which is not isolated (*Cleaning the tank*) - will be true.

3.6 Validation, Preservation and Guarantee Algorithms

After specifying a constraint, we can apply it to an individual step or to the whole transformation. The pseudo codes of the algorithms are the following:

```
VALIDATION (VMTSConstraint const, VMTSControl control,
VMTSMatch match): bool
 1 VMTSRule firstRule = control.getFirstRule()
 2 VMTSRule lastRule = control.getLastRule()
 3 ADD_PRECONDITION_TO_RULE(firstRule, const)
 4 ADD_POSTCONDITION_TO_RULE(lastRule, const)
 5 return EXECUTE_CONTROL(control, match)
```

Three parameters are passed to the validation algorithm, the first parameter is a *VMTSConstraint* which contains the constraint we want to validate, the second one is a *VMTSControl*, which contains the rewriting rules in an ordered sequence, and the third one is a *VMTSMatch* which contains the matched nodes. The validation algorithm selects the first and the last rules from the control, and enlists the given constraint as a precondition in the selected first step and as a postcondition in the selected last step. This will be proper if the control contains only one step, and also if the control contains a finite sequence of steps, because if there is only one step in the control, the *getFirstRule* and the *getLastRule* queries return the same step. Finally the algorithm calls the EXECUTE_CONTROL method and retrieves its return value.

The second pseudo code describes the EXECUTE_CONTROL method, which is used by all three (validation, preservation, guarantee) algorithms.

```
EXECUTE_CONTROL (VMTSControl control, VMTSMatch match): bool
 1 foreach VMTSRule rule of control.getRulesInOrderedSequence()
 2   if rule.hasPrecondition and not (CHECK_PRECONDITIONS(rule, match))
 3     then return false
 4   end if
 5   FIRE_RULE(rule, match)
 6   if rule.hasPostcondition and not (CHECK_POSTCONDITIONS(rule, match))
 7     then return false
 8   end if
 9 end foreach
10 return true
```

The EXECUTE_CONTROL method applies the rewriting rules contained by the given control to the passed match. If a step has a precondition, the methods calls the CHECK_PRECONDITIONS method, and if it returns false, then the execution of the step and the whole finite sequence of steps fails, and the algorithm returns false. Otherwise the method calls the FIRE_RULE function, and after that if the rule has a postcondition then the procedure is similar to the case of preconditions.

```
PRESERVATION (VMTSConstraint const, VMTSControl control,
VMTSMatch match): bool
 1 VMTSRule firstRule = control.getFirstRule()
 2 VMTSRule lastRule = control.getLastRule()
 3 ADD_POSTCONDITION_TO_RULE(lastRule, (NOT  const ||
   const@pre_firstRule) && (const || NOT  const@pre_firstRule))
 4 return EXECUTE_CONTROL(control, match)
```

The preservation algorithm selects the first and the last rules from the control, creates a constraint expression based on the given constraint: $(NOT\ P\ or\ P@preS_1)\ and\ (P\ or\ NOT\ P@preS_1)$, enlists this created expression as an OCL expression in the postconditions of the selected last step, and finally calls the EXECUTE_CONTROL method and obtains its return value.

```
GUARANTEE (VMTSConstraint const, VMTSControl control, VMTSMatch match,
int indexOfMarked): bool
 1 if (control.Rules.Length == 1 || indexOfMarked < 0)
 2   then ADD_POSTCONDITION_TO_RULE(control getLastRule(), const)
 3 else
 4   ADD_POSTCONDITION_TO_RULE(control.getRuleByIndex(indexOfMarked),
     const)
 5   for i = indexOfMarked + 1 to control.Rules.Length
 6     ADD_PRECONDITION_TO_RULE(control.getRuleByIndex(i), const)
 7     ADD_POSTCONDITION_TO_RULE(control.getRuleByIndex(i), const)
 8   end for
 9 end if
10 return EXECUTE_CONTROL(control, match)
```

The fourth parameter of the guarantee algorithm - $indexOfMarked$ - contains the index of a marked step, this step guarantees the given property, and the steps after this marked step preserve this property. If the given control has exactly one step, or if the given $indexOfMarked$ is less than zero (there is no marked step), the algorithm enlists the given constraint in the postconditions of the last step. If the parameter $indexOfMarked$ contains valid index, the algorithm enlists the given constraint in the postconditions of the step specified by the $indexOfMarked$, and enlists the constraint in both the pre- and the postconditions of the steps $S_{indexOfMarked+1}, \ldots S_n$. Finally the algorithm calls the EXECUTE_CONTROL method and retrieves its return value.

4 Conclusions

Our metamodel-based specification of the rules allows assigning OCL constraints to the rules, and they are able to express local constraints. However, it does not mean that validating them does not involve checking other model elements in the input model. OCL constraints enlisted in the rules have effect on the instances of these rules, on the matched and the replaced subgraphs.

We have shown the relationship between the pre- and postconditions and OCL constraints, and how we can use the OCL constraints enlisted in the rewriting rules to check validation, preservation and guarantee properties, or simply how to check models with the help of metamodel-based graph rewriting. If the developer is familiar with UML, then this method seems quite natural, hence the UML knowledge can be transferred with a little justification.

In this paper the concepts of general validation, general preservation and general guarantee have been discussed. It is presented that if a rewriting rule (a finite sequence of steps) specified adequately with the help of constraints, and the rule (the finite sequence of steps) has been executed successfully for the input model, then the generated output model is in accordance with the expected result, which is described by the rewriting rule (the finite sequence of steps) refined with the constraints. It means that the modeler's task is to create adequate rules and specify them fully with constraints (pre- and postconditions), and if the execution of the rewriting rule (finite sequence of steps) finishes successfully, it produces a valid result.

The main limitation of the presented method is the local-nature of the rules. If one wants to specify a constraint for an element, (i) it must be included in a rewriting rule, or (ii) it must be referenced by the OCL traversal expressions assigned to the rule elements. Consequently, this method does not provide an easy way to check global constraints such as dead lock examination. For those parts of the graphs that are not affected by a rewriting rule we can not specify constraints. But there are numerous cases such as the elaborated code generation from statechart model, where the whole right side is generated, thus all the output model elements can be validated.

References

[1] D. Akehurst, O. Patrascoiu, *OCL 2.0 - Implementing the Standard for Multiple Metamodels*, Workshop Proceedings, 6th International Conference on the Unified Modeling Language and its Applications, <<UML>>2003, Electronic Notes in Theoretical Computer Science (ENTCS), October 2003.

[2] D. Blostein, H. Fahmy, A. Grbavec, *Practical Use of Graph Rewriting*, Technical Report No. 95-373, Department of Computing and Information Science, Queen's University, Kingston, Ontario, Canada, January, 1995.

[3] Corradini A, Ehrig H, Löwe M, Montanari U, Rossi F, *Abstract Graph Derivations in the DoublePushout Approach.* In H.J. Schneider and H. Ehrig, editors, Proceedings of the Dagstuhl Seminar 9301 on Graph Transformations in Computer Science, vol.776 of Lecture Notes in Computer Science, pp. 86–103. Springer Verlag, 1994.

[4] Ehrig H, *Introduction to the Algebraic Theory of Graph Grammars*, In Graph Grammars and Their Applications to Computer Science and Biology, Springer, Ed. Claus V., Ehrig H., Rozemberg G., Berlin, 1979.

[5] Ehrig H, *Tutorial introduction to the algebraic approach of graph grammars.* In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, Proceedings of the 3rd International Workshop on GraphGrammars and Their Application to Computer Science, volume 291 of Lecture Notes in Computer Science, pages 3–14. Springer Verlag, 1987.

[6] Ehrig H, Korff M, Löwe M, *Tutorial introduction to the algebraic approach of graph grammars based on double and single pushouts.* In H. Ehrig, H.J. Kreowski, and G. Rozenberg, editors, Proceedings of the 4th International Workshop on GraphGrammars and Their Application to Computer Science, volume 532 of Lecture Notes in Computer Science, pages 24–37. Springer Verlag, 1991.

[7] Ehrig H (ed.), *Handbook on Graph Grammars and Computing by Graph Transformation: Foundations*, Vol. 2. World Scientific, Singapore, 1997.

[8] Ehrig H, Engels G, Kreowski H-J, Rozemberg (ed.), *Handbook on Graph Grammars and Computing by Graph Transformation: Application, Languages and Tools*, Vol.2. World Scientific, Singapore, 1999.

[9] Generic Modeling Environment (GME) 2000, http://www.isis.vanderbilt.edu/Projects/gme/default.html

[10] Ali Hamie, John Howse, Stuart Kent, *Interpreting the Object Constraint Language*, Proceedings 5th Asia Pacific Software Engineering Conference (APSEC '98), December 2-4, 1998, Taipei, Taiwan, 1998.

[11] Karsai G, Agrawal A, Shi F, Sprinkle J, *On the Use of Graph Transformation in the Formal Specification of Model Interpreters*, Journal of Universal Computer Science, Special issue on Formal Specification of CBS, 2003.

[12] Levendovszky T, Lengyel L, Charaf H, *Implementing a Metamodel-Based Model Transformation System*, Buletinul Stiintific al Universitatii Politehnica din Timisoara, ROMANIA Seria AUTOMATICA si CALCULATOARE PERIODICA POLITECHNICA, Transactions on AUTOMATIC CONTROL and COMPUTER SCIENCE Vol.49 (63), 2004, ISSN 1224-600X.

[13] Levendovszky T, Lengyel L, Mezei G, Charaf H, *A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS*, International Workshop on Graph-Based Tools (GraBaTs) Electronic Notes in Theoretical Computer Science, Rome, 2004.

[14] Levendovszky T, Lengyel L, Charaf H, *Software Composition with a Multipurpose Modeling and Model Transformation Framework*, IASTED 2004, Innsbruck, 2004, pp.590-594.

[15] Sten Loecher, Stefan Ocke, *A Metamodel-Based OCL-Compiler for UML and MOF. In OCL 2.0 - Industry standard or scientific playground*, Workshop Proceedings, 6th International Conference on the Unified Modeling Language and

its Applications, <<UML>>2003, Electronic Notes in Theoretical Computer Science (ENTCS) , October 2003.

[16] B. Meyer: *Object-Oriented Software Construction*, Prentice Hall, New York, 1988.

[17] MDA Guide Version 1.0.1, OMG, document number: omg/2003-06-01, 12th June 2003 www.omg.org/docs/omg/03-06-01.pdf

[18] Object Constraint Language Specification (OCL), www.omg.org

[19] UML 2.0 Specifications, http://www.omg.org/uml/

[20] PROGRES system can be downloaded from http://www-i3.informatik.rwth-aachen.de/research/projects/progres/main.html

[21] G. Rozenberg (ed.), *Handbook on Graph Grammars and Computing by Graph-Transformation: Foundations*, Vol.1 World Scientific, Singapore, 1997.

[22] Sprinkle J: *Notes for Model-Integrated Computing*, EECE290O, Berkeley, http://www.eecs.berkeley.edu/ sprinkle/teaching/eecs290o/reading/book/

[23] Sztipanovits J, Karsai G, *Model-Integrated Computing*, IEEE Computer, pp. 110–112, April, 1997.

[24] Sztipanovits J, Karsai G, *Generative Programming for Embedded Systems*, LNCS 2487, pp. 32–49, 2002.

[25] Visual Modeling and Transformation System Web Site http://avalon.aut.bme.hu/ tihamer/research/vmts/

Acta Cybernetica **17** (2005) 359–383.

Reconstruction of binary matrices from fan-beam projections

Antal Nagy* and Attila Kuba*

Abstract

The problem of the reconstruction of binary matrices from their fan-beam projections is investigated here. A fan-beam projection model is implemented and afterwards employed in systematic experiments to determine the optimal parameter values for a data acquisition and reconstruction algorithm. The fan-beam model, the reconstruction algorithm which uses the optimization method of Simulated Annealing, the simulation experiments, and the results are then discussed in turn.

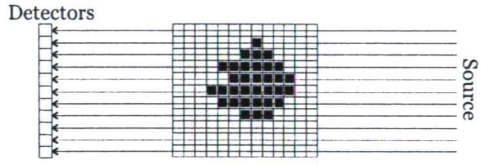
1 Introduction

Tomography is an imaging procedure where the cross-sections of the 3D object being studied are determined from its projection images. The projection images can be created by some rays that are emitted from a source (like X-rays from an X-ray tube), transmitted through and partially absorbed by the object, and finally detected by some array (plane or line) of detectors. The pixels of the projection image represent the total absorption of the rays along the lines between the source and the corresponding detector elements. Usually several projections of the object are acquired from different directions. Then the task is to compute the cross-sections of the object via some mathematical procedure [1] called *reconstruction from projections*. This imaging technique is routinely used in *computerized tomography* (CT) for example, where the section images of the human body are computed from a huge number of measurements using transmitted X-rays. The general method for the reconstruction of 3D objects is that the 2D cross-sections of the objects are reconstructed from the projections measured in the plane of the selected section, effectively reducing the 3D reconstruction problem to a series of *2D reconstruction problems*. In the case of so-called *truly 3D reconstruction* the whole 3D object is reconstructed using rays in the whole 3D space.

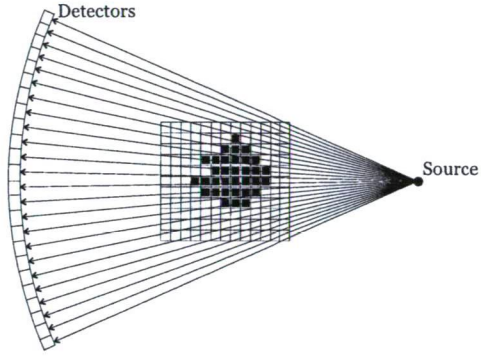
Discrete tomography (DT) is a special kind of tomography that can be applied if the object to be reconstructed consists of only a few known homogeneous materials

*Department of Image Processing and Computer Graphics, University of Szeged, Szeged, Hungary, Email: nagya@inf.u-szeged.hu

(e.g. metal and wood). This information can be incorporated into the reconstruction process, giving one the opportunity of reconstructing simple objects from a much smaller number of projection values than is necessary for more complex objects. For this reason discrete tomography seems to be important in applications where the object is so simple and there is no opportunity or it is too costly to acquire lots of projections, like those in non-destructive testing, electron microscopy and medicine. For a summary of the theory and applications of DT, see [2].



(a) Parallel projections



(b) Fan-beam projections

Figure 1: Parallel and fan-beam projections.

There are basically two ways of acquiring the necessary projections (see Fig. 1). In the case of *parallel projections*, the rays parallel to a given direction are transmitted and measured in one phase of the acquisition process. By rotating the system other rays parallel to other directions can be created. In the case of *fan-beam projections* the rays coming from the actual source position (like a fan) are measured at each step. By rotating the source and the detectors around the object new fan-beam projections can be created. Although the two kinds of projection are really equivalent from the viewpoint of information available for the reconstruction (by a suitable rearrangement of the rays one can be transformed to the other), there are different reconstruction methods for parallel and fan-beam projections. This type of classification is common in applications where, owing to the technical possibilities, one of these arrangements is always used.

In this paper we discuss a special discrete tomography problem, namely the reconstruction of binary matrices from their fan-beam projections. The reconstruc-

tion of binary matrices from parallel projections is a classical problem and it has been intensively investigated (see [3, 4], for example). Even the reconstruction from fan-beam (in 2D) or from cone-beam (in 3D) projections is well understood (see [5, 6]). It is interesting that at the same time there are very few papers about DT using fan-beam/cone-beam projections. To our knowledge, the only paper to date that deals with this problem is the article by Peyrin *et al.* [7]. There they discuss results related to truly 3D reconstructions of objects from cone-beam projections. The main reason for this might be that from a mathematical viewpoint some reconstruction results of parallel projections can be applied directly in the case of fan-beam projections, but this is not the case when we desire, for example, the optimal number of X-ray sources, the necessary minimum number of measurements, and so on.

We believe that 2D reconstruction using fan-beam rays is an important and interesting problem especially from an application point of view.

There are several applications of tomography that make use of fan-beam projections and could be of interest in DT e.g. non-destructive testing using X-rays [8] or neutron beams [9].

The aim of this paper is to investigate the quality of reconstruction as a function of the parameters of the fan-beam projection model and the reconstruction algorithm applied. The method employed is the simulation of the whole process from acquiring projections to a comparison of the reconstructed images. The whole simulation procedure is realized in the following way. Binary matrices are created which represent the 2D objects to be reconstructed. Then a model for computing the fan-beam projections is set up and implemented. The fan-beam model contains several parameters like the number of sources and detector elements. With suitable parameter settings different fan-beam data acquisition systems can be simulated. The projections are afterwards computed analytically based on the parameter values of the fan-beam model. The measurement errors can be simulated in our system by some additive random noise. A random-search optimization method was implemented here to reconstruct binary images from the input data. In order to compare the reconstructed images with the original image in an objective way, several measures are implemented; here we present the results of this comparison expressed as a relative mean error. The effects of each parameter are studied in such a way that a sequence of reconstructions is performed by varying only one parameter value and keeping the others fixed. In this way we can produce a curve of the values for the relative mean square and show the effect of the given parameter on the reconstruction process.

The structure of the paper is as follows. In Section 2 the reconstruction problem is introduced with the necessary definitions and our notation for fan-beam projections. Afterwards, we describe the details of the fan-beam model used in our simulation experiments. In Section 4 our DT reconstruction problem is reformulated as an optimization problem, which is then solved using the method of Simulated Annealing (SA). The results of our experiments together with related discussions are given in Section 5. Finally, in the last section, we present conclusions from the studies carried out so far and make suggestions for future work.

2 The reconstruction problem for fan-beam projections

Let f be an integrable real function in the $\mathbb{R}^2$ plane. Let S be a point called the *source point*, and v_θ be a unit vector in the direction $\theta \in [0, 2\pi)$ in the plane. Consider the integrals of f along the half-lines starting from S in direction v_θ

$$[\mathcal{R}f](S,\theta) = \int_0^\infty f(S + u \cdot v_\theta)du \ . \tag{1}$$

The transformation defined by (1) is called the *projection of f taken from the point S in the direction θ*, or the *fan-beam projection of f taken from the point S*. (Another way of acquiring projections is when the integrals of f are taken along parallel straight lines in given directions. This kind of projection is called *parallel*.) Such a projection model can be applied to computerized tomography (see [1], say) where the projections are taken from several hundred source points around the object to be reconstructed.

Given a set of the source points $\mathcal{S}$, the *reconstruction problem using fan-beam projections* can be stated as follows:

FB($\mathcal{S}$) RECONSTRUCTION PROBLEM

Given: *A function $g : \mathcal{S} \times [0, 2\pi) \longrightarrow \mathbb{R}$.*

Task: *Construct a function f such that*

$$[\mathcal{R}f](S,\theta) \ = \ g(S,\theta)$$

for all $S \in \mathcal{S}$ for almost every $\theta \in [0, 2\pi)$.

There are several methods for solving the FB reconstruction problem. For a summary the interested reader may read [1], for example.

In this paper we are interested in the reconstruction of special types of functions from fan-beam projections. Henceforth, let us suppose that the support of f can be covered by an $n \times n$ regular lattice W such that f is constant on each 1×1 square of the lattice, so f can take a value 0 or 1. That is, f can be represented by a binary-valued matrix or, equivalently, by a vector $\mathbf{x} \in \{0,1\}^J$ where x_j denotes the jth element of the matrix, say, in successive order, where $j = 0, 1, \ldots, J$ and $J = n^2$.

In the majority of applications the projections are acquired from only a finite number of points, S_k, $k = 1, 2, \ldots, K$, along a finite (L) number of half-lines from each point. In this case the ith projection, b_i, from the point S_k in direction v_l ($i = (k-1) \cdot K + l$) can be described by the linear equation

$$\sum_{j=0}^{J} a_{ij} x_j \ = \ b_i \,, \qquad i = 1, 2, \ldots, I \ , \tag{2}$$

where a_{ij} denotes the length of the intersection of the ith half-line with the jth unit square of W and $I = K \cdot L$. In the linear equation system (2) the projections are obtained (within a certain error) by measurements. The elements of matrix $A = (a_{ij})_{I \times J}$ can be computed knowing the positions of the squares in W and the half-lines starting from the source points. The special feature of (2) is that the unknown vector $\mathbf{x}$ is binary here, i.e. $x_j \in \{0, 1\}$ for all $j = 1, 2, \ldots, J$.

3 The fan-beam model

Based on the method of data acquisition in most of the DT applications, as in our fan-beam model (see Fig. 2), the source points S_k, $k = 1, 2, \ldots, K$, lie on a circle $C_r = \{(x, y) \mid x^2 + y^2 = r^2\}$ around the origin O, where $r > 0$ is large enough for W to be in C_r. Furthermore, it is also usual that the source points are uniformly distributed on C_r, that is $S_k = (r \cdot \cos\theta_k, r \cdot \sin\theta_k)$, where $\theta_k = \theta_0 + (k-1) \cdot 2\pi/K$ for all $k = 1, 2, \ldots, K$. The *start angle* $\theta_0 \in [0, 2\pi)$ determines not only the position of the first point, but all source points. (The reason for the inclusion of the start angle θ_0 into the model is that since we usually have only a few source points (like 2–4), their number as well as their positions can have a strong influence on the reconstruction, as we shall show later.) For example, a start angle of 0° means that the first source lies on the intersection of circle C_r and also on the positive part of the x axis.

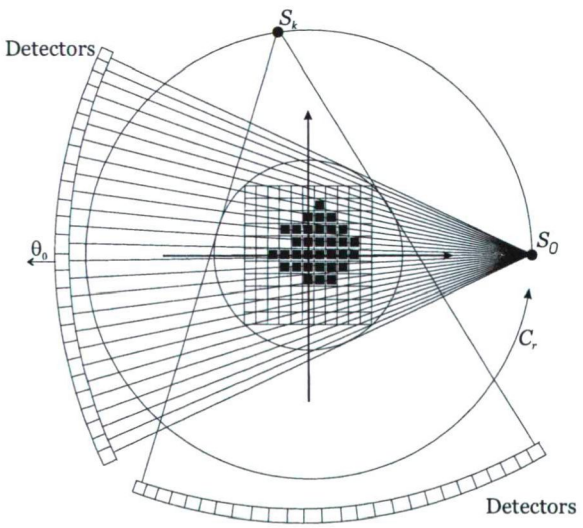


Figure 2: The geometry of our fan-beam model.

In our model the integrals along the half-lines starting from the source point S_k ($k = 1, 2, \ldots, K$) are measured by L detectors, uniformly placed on the other

side of the object on an arc having its center point in S_k (see Fig. 2). The arc of detectors is big enough for the whole image to lie between the half-lines drawn from the source to the endpoints of the detector arc. Each detector measures one projection value b_i.

For simplicity, we suppose that the center of the rectangle W is at the origin of the coordinate system (see Fig. 2).

We are going to study the effect of noise as well. That is why Gaussian noise was generated and added to the exact projections to create noisy projection data.

In our fan-beam model the following parameters can be varied:

n: the size of the binary matrix to be reconstructed ($n \times n$);

r: the radius of circle C_r, i.e. the distance of the source points from the origin;

θ_0: the start angle which determines the position of the first source point;

K: the number of source points;

L: the number of detector elements or, equivalently, the number of measurements from one source point;

η: the percentage of Gaussian noise added in the projections.

4 Reconstruction as an optimization problem

As we saw earlier the solution of the FB (see Section 2) reconstruction problem in our fan-beam model is equivalent to finding a solution of the linear equation system

$$\mathbf{Ax} = \mathbf{b}, \qquad \text{where } \mathbf{x} \text{ is a binary-valued vector}. \tag{3}$$

Since any half-line in our model at most intersects $O(n)$ squares of W, the matrix $\mathbf{A} = (a_{ij})_{I \times J}$ is sparse (it contains only a few non-zero elements). Another important property of this matrix equation in DT applications is that the number of equations (i.e. the number of projections) is usually much less than the number of unknowns, hence $I << J$. It means that it can have several solutions, even binary-valued ones. Furthermore, due to measurement errors it is also possible that (3) has no exact solution, so it is better to try to find a binary-valued $\mathbf{x}$ which satisfies (3), at least approximately.

The reconstruction methods like the Algebraic Reconstruction Techniques (ART) (see [1], say) do not necessarily provide a binary-valued $\mathbf{x}$ that satisfies (2). It cannot be applied here because a non-binary solution might be quite different from a binary one.

Actually, a possible way of solving (3) at least approximately is to reformulate it as an optimization problem. Formally, we should find the minimum of the following objective function

$$C(\mathbf{x}) = |\mathbf{Ax} - \mathbf{b}| + \gamma \cdot \Phi(\mathbf{x}), \qquad \text{where } \mathbf{x} \text{ is a binary-valued vector}. \tag{4}$$

The first term on the rhs ensures that we have an $\mathbf{x}$ satisfying (3) at least approximately. The second term allows us to include *a priori* knowledge about $\mathbf{x}$ into the optimization if there are several good binary vector candidates that keep $|\mathbf{Ax} - \mathbf{b}|$ low. For example, $\Phi(\mathbf{x}) = |\mathbf{x} - \mathbf{x}^{(0)}|$ might be such a case, where $\mathbf{x}^{(0)}$ is a given (probably binary-valued) vector, called a *prototype*. With such a Φ we can look for an $\mathbf{x}$ that is similar to the prototype $\mathbf{x}^{(0)}$. The regularization coefficient γ is needed to weight the two terms in C. For example, if the measurements are very noisy then we can choose a bigger γ than that for the noiseless case to give the second term in (4) a higher weight, so it will play a more important role in the optimization. γ can also be used to control the effects of noise.

Since we are looking for a binary-valued $\mathbf{x}$ in the optimization of (4), the usual numerical optimization methods seem unsuitable here. The combinatorial optimization methods looked more promising and turned out to be useful. Among them we selected the *simulated annealing* (SA) optimization procedure [10]. The reason for this selection was that it was easy to implement; it can be easily adapted to the objective function (i.e. with very small modifications the program is suitable for optimizing other objective functions). Besides, since our main aim is to study the effects of the fan-beam model parameters, the selection of the optimization method actually plays only a secondary role here. We believe that we would probably get similar results with other optimization methods because they have to optimize the objective function as well.

4.1 Simulated annealing

SA is a random-search technique that is based on the physical phenomenon of metal cooling. The system of metal particles gradually reaches the minimum energy level where the metal freezes into a crystalline structure. Based on previous works [11], we implemented the SA algorithm like so (see Fig. 3):

The algorithm starts from an arbitrary initial binary image $\mathbf{x}^{(0)}$, an initial (high) temperature $T^{(0)}$ and calculates the objective function value $C(\mathbf{x})$. Then a position j is randomly chosen in the image $\mathbf{x}$. Let $\mathbf{x}'$ be the image that differs from $\mathbf{x}$ only by changing the value of $\mathbf{x}$ in position j to the other binary value, i.e. $x'_j = 1 - x_j$. This change is accepted by the algorithm, i.e. $\mathbf{x}$ is replaced by $\mathbf{x}'$ if $C(\mathbf{x}') < C(\mathbf{x})$. Even if the objective function does not get smaller, the change is accepted with a probability depending on the difference $\Delta C = C(\mathbf{x}') - C(\mathbf{x})$. Formally, the change is accepted even in that case when

$$\exp(-\Delta C/\kappa T) > z\,, \tag{5}$$

where κ, T and z are, respectively, the Boltzmann constant ($11.3805 \times 10^{-23} \times m^2 kg s^{-2} K^{-1}$), current temperature and a randomly generated number from a uniform distribution in the interval $[0, 1]$. Otherwise, the change is rejected, i.e. $\mathbf{x}$ does not change in this iteration step.

If a change is rejected then we test the level of *efficiency* of changes in the image in the last iterations. It means that we count the number of rejections in

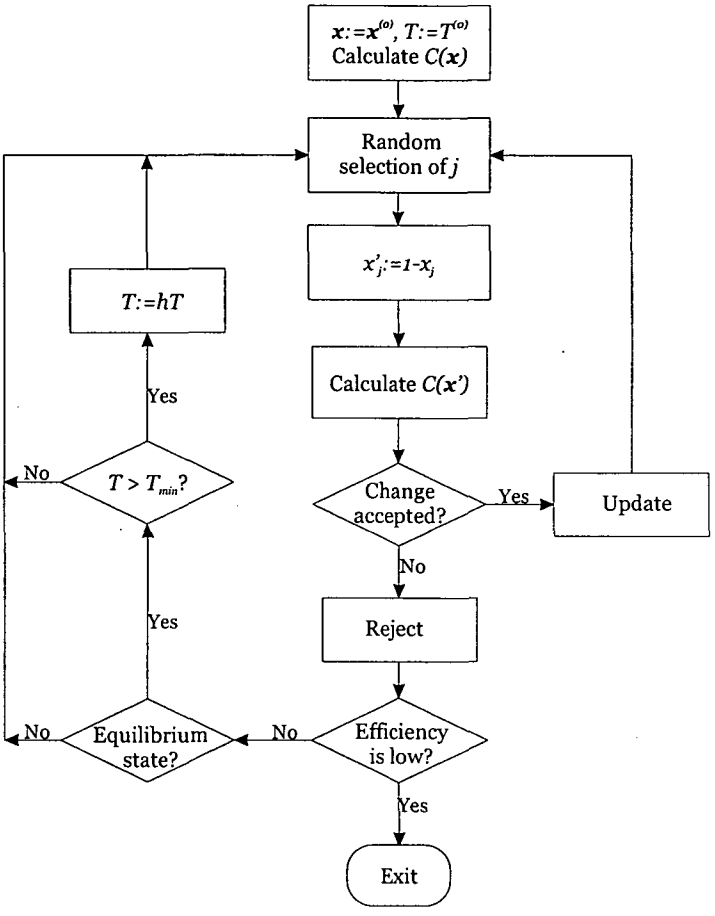


Figure 3: Flow-chart of the implemented SA algorithm.

the last N_{iter} iterations. If this number is greater than a given threshold value R_{thr} then the efficiency of changes is too low and the SA optimization algorithm will be terminated.

We calculate the variance of the cost function in the last N_{var} iterations. A so-called *equilibrium state* is said to be attained if the present estimate of the current ΔC variance is greater than the previous variance estimate. If the equilibrium state is achieved, we reduce the current temperature (allowing changes with smaller probabilities when the value of the objective function is greater) and let the algorithm run with a lower temperature value (T is replaced by $h \cdot T$, where h is the so-called *cooling factor*). In our experiments we chose the same value for the parameter as in [7], namely $h = 0.9$.

Our SA algorithm then has the following parameters:

$\mathbf{x}^{(0)}$: initial image;

$T^{(0)}$: initial temperature;

N_{iter}: number of iterations used in the computation of efficiency;

R_{thr}: threshold value for the number of rejected changes in the last N_{iter} iterations;

N_{var}: number of accepted iterations used in the computation of the variance of the cost function.

5 Results and discussion

In this section the experimental results using our fan-beam model and the implemented SA algorithm are presented together with a discussion of each.

The simulation experiments were performed with phantom images each having a size 200×200 (i.e. $n = 200$). The projections of the phantom images were computed based on (2) for each parameter setting. The images were then reconstructed from the projections using the SA algorithm outlined above. In order to get quantitative results, the original phantom images were compared pixel by pixel according to the relative mean error

$$M_e = \frac{\sum_{j=1}^{J} |x_j - \hat{x}_j|}{\sum_{j=1}^{J} \hat{x}_j}, \tag{6}$$

where $\hat{\mathbf{x}} = \{\hat{x}_j\}_{j=1}^{J}$ denotes the vector of the original image. Clearly, $M_e \geq 0$ and the smaller value indicates a better comparison result. Furthermore, $M_e = 0$ if and only if $\mathbf{x} = \hat{\mathbf{x}}$.

Since we had an optimization process based on a random-search, we repeated each test 100 times with the same parameter setting. The mean of the 100 M_e values was computed and presented later as the result for each test with the given parameter setting. The average image of the 100 binary images was given as the result of the reconstruction for one parameter setting.

Naturally, several parameter settings were tested. One of them, the so-called *baseline parameter setting*, played a special role. Here only one of the parameters was allowed to change at a time, the others having the same values as in the baseline parameter setting case. In order to see the effect of the parameters on the quality of the reconstruction, a sequence of tests was performed for each parameter. During a test sequence only the value of the selected parameter was changed and the other parameters always had the same values as in the baseline parameter setting case. For instance, to see the effect of increasing the number of sources, we changed the value of K in the model (lying in the range 2–32), computed the projections for the same phantom image, ran the reconstruction algorithm with the

Table 1: Baseline parameter setting

Parameter	Baseline value	Range
Distance from sources to origin (r)	250	$[250, 1750]$
First source position angle (θ_0)	$0°$	$[0°, 360°/K]$
Number of sources (K)	$K \in \{22, 32\}$	$[2, 32]$
Number of detector elements (L)	401	$[101, 401]$
Additive noise	$\eta \in \{0\%, 5\%\}$	$[0\%, 45\%]$
Initial temperature (T)	$4°$	$[4°, 104°]$
Number of iterations (N_{iter})	10000	$[1000, 10000]$
Rejected iterations (R_{thr})	9990	$[990, 9990]$
Variance iterations (N_{var})	5000	$[500, 5000]$
Regularization parameter (γ)	145	$[0, 145]$

same parameter settings 100 times, took the reconstructed 100 images, computed the 100 M_e values for the reconstructed and original images, calculated the average images, then finally drew a curve showing the changes of M_e as a function of the number of projections. The curve drawn from the mean values of a sequence like this is presented as the final result of the observations associated with the chosen parameter. The baseline parameter setting together with the range of the parameter values are given in Table 1.

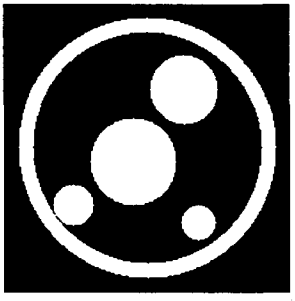


Figure 4: Baseline software phantom used in the tests.

The experiments were executed with the Φ_{poz} penalty term and $\gamma = 145$ regularization parameter (Section 5.4).

As can be seen from Table 1, the noise contribution was only 0% or 5%. In order to assess the effect of noise we repeated the tests not just with 0% noise but also with 5% noise. From the test results, we obtained two curves, one without noise and one with 5% noise.

We repeated the test for $K = 32$ and $K = 22$ in most of the experiments. These produced two more curves for the experiments that were performed.

The parameters and the corresponding results can be divided into four groups. This is why we have chosen to discuss the results separately in the subsequent

four subsections. They show the effect of changing the parameters of the fan-beam model, the SA optimization algorithm, the complexity of the phantom image to be reconstructed, the regularization parameter γ given in (4), and the effect of adding noise.

5.1 The parameters of the fan-beam model

In this group of tests we studied the effects of changing those parameters related to the fan-beam model listed in Section 3.

5.1.1 Distance between sources and origin

This parameter was varied between 250 and 1750 while the detector angular aperture covered the W lattice to be reconstructed and the number of detectors L were kept constant (see Fig. 5). Of course, as the distance between the source points and the origin increases, the fan-beam model approaches the model of parallel projections when the same detector parameters are used.

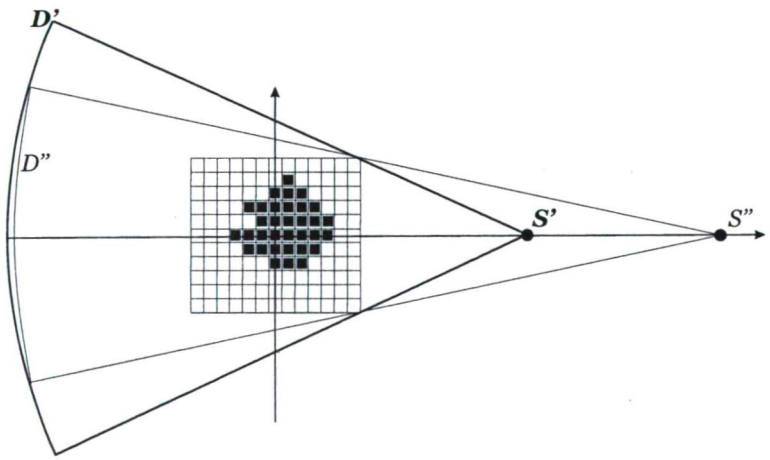
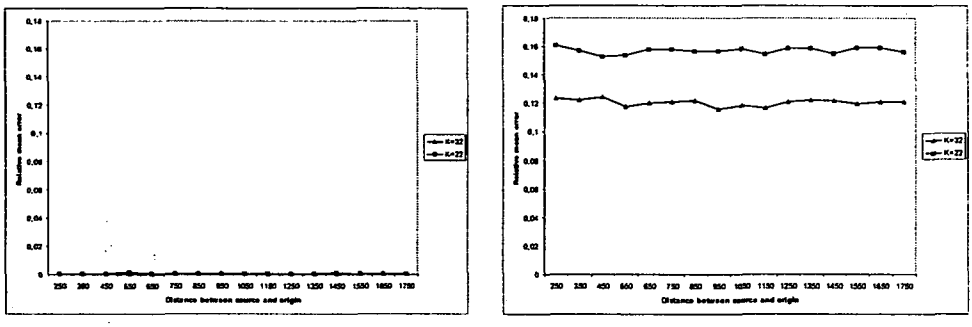


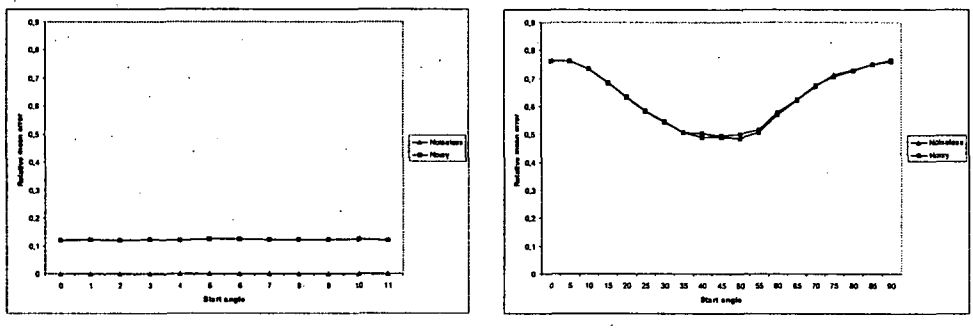
Figure 5: Changing the distance between source and origin. Detector arc D' is for the source point S' and detector arc $D"$ is for the source point $S"$.

The curves in Fig. 6 both show that there is no big difference between the results when the source is close to or far from the origin. More generally, there is no real difference between the fan-beam and parallel-beam projections if we change this distance. This is primarily because the further we go from the origin, the more parallel the beams become. It seems that the equations belonging to the near-parallel rays determine the image in both cases.



(a) No noise (b) 5% Gaussian noise

Figure 6: Relative mean error as a function of the distance between source and origin.



(a) 32 projections (b) 4 projections

Figure 7: Relative mean error as a function of the start angle.

5.1.2 Start angle

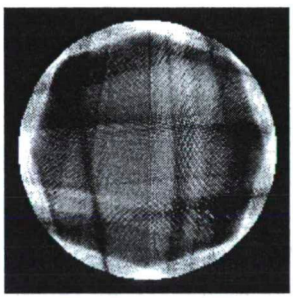
We varied the value of the start angle parameter from 0° to $360°/K$ degrees. It is clear that by determining the position of the first source we also determine the positions of all sources around the circle C_r (if the number of sources is fixed). The relative mean square curves describing the effects of these changes are given in Fig. 7(a) for the case when the number of sources is just the base setting (i.e. $K = 32$). The curves indicate that there is little real difference in the relative mean square if we have a relatively large number of source points.

But we obtain quite different curves when the number of projections is small. For example, for $K = 4$ source points we get curves which show that the quality of the reconstruction changes depends on the source positions (see Fig. 7(b)).

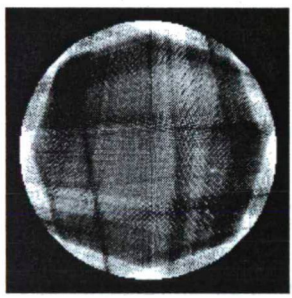
The reason for the shape of the curves in Figs. 7(a) and 7(b) is that there are certain projection values which provide more information for the reconstruction

than others. For example, a projection value of 0° means that along the corresponding half-line all pixels intersecting the half-line have a value of 0° (which may be called an *empty line of pixels*). Another is when a projection value is the same as the length of the intersection of the square in W with the corresponding half-line; in that case all pixels along the half-line have a value 1 (let us call them a *full line of pixels*). If the source points have positions such that there are many pixels lying in almost empty or almost full lines then a large part of the image can be reconstructed from a few projections. Looking at Fig. 4 we notice that this is indeed the case when one of the source points is in a position near 40°.

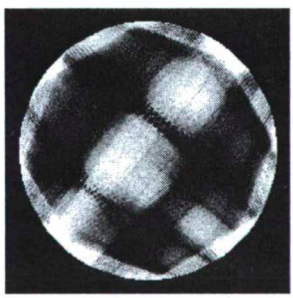
The reconstructed images can be seen in Fig. 8 when the number of sources is 4. These figures confirm the earlier belief that the quality will be better when the first source point lies almost on the same line as the centers of 3 circles.



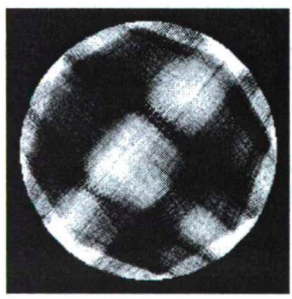
(a) Start angle = 0°, no noise



(b) Start angle = 0°, 5% Gaussian noise



(c) Start angle = 40°, no noise

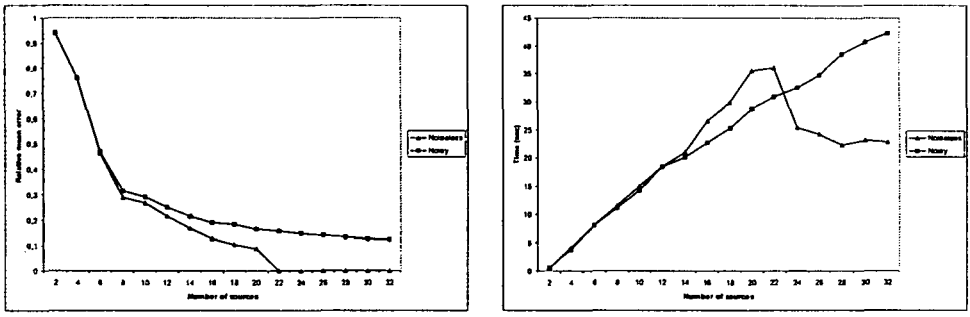


(d) Start angle = 40°, 5% Gaussian noise

Figure 8: Images reconstructed from 4 projections with different start angles.

5.1.3 Number of sources

The number of sources in the simualtions was varied from 2 to 32. It came as no surprise that increasing the number of sources and number of projection values in



(a) Relative mean error (b) Time (sec)

Figure 9: The effect of varying the number of sources.

turn increased the reconstruction quality. But the real question here was how to find the minimum number needed to produce a good reconstructed image for a particular case. Figure 9 provides the information needed to answer this question. It is also tells us what this number depends on.

The graphs in Fig. 9(a) show that taking more than 12 sources improves the mean square error by a very small amount. When there were noiseless projections the data from 22 sources was sufficient to achieve high quality reconstructions.

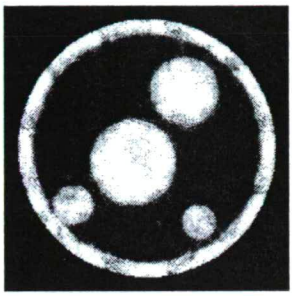
The view, based on the graphs in Fig. 9(a), that 22 or more source points hardly affects the quality of the reconstruction is indeed borne out by inspecting the reconstructed images (Fig. 10). At the same time reconstructing the phantom from 22 projections takes more time than reconstructing it from 32 projections when no Gaussian noise was added (see Fig. 9(b)).

5.1.4 Number of detector elements

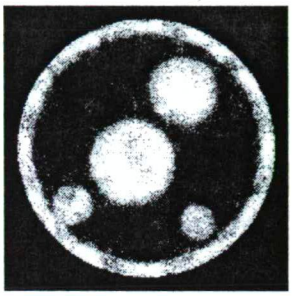
The number of detector elements also means that the number of projection values measured will belong to a source point. If we have more detector elements we will also have more equations in (2), hence more information about the image. But, of course, for the detector elements beyond a certain number we cannot obtain a better mean square error (Fig. 11).

That increasing the number of detector elements will bring an improvement only up to some limit can be readily explained. Here it is about 281 when $K = 32$ (see Figs. 11 and 12) and the binary image seems to be determined by the exact projection data.

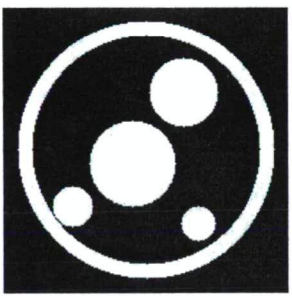
This experiment was repeated with $K = 22$. We got approximately the same relative mean error level for $L = 401$ when no noise was added (see Fig. 11(a)). The number of equations to be solved is almost the same in both cases (i.e. $32 \cdot 281 \approx 22 \cdot 401$), which helps to explain why the results were so similar.



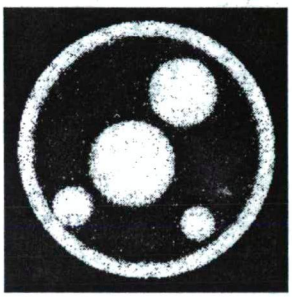
(a) 12 sources, no noise



(b) 12 sources, 5% Gaussian noise

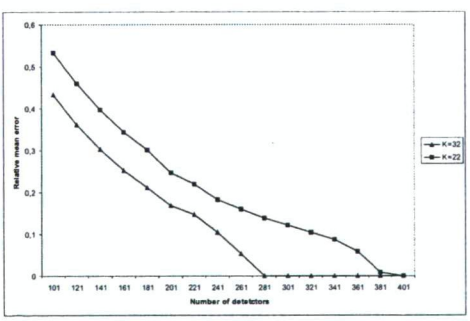


(c) 22 sources, no noise

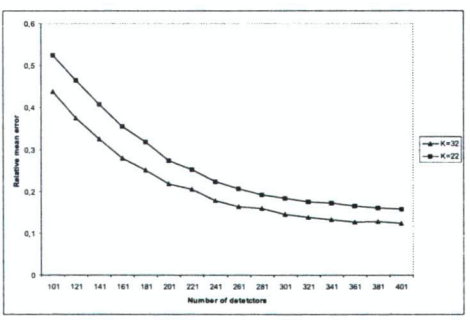


(d) 22 sources, 5% Gaussian noise

Figure 10: Images reconstructed from projections with a different number of sources.

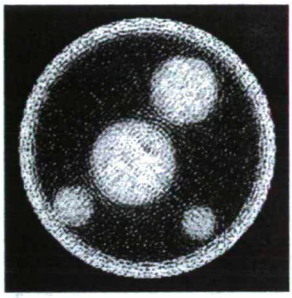


(a) No noise

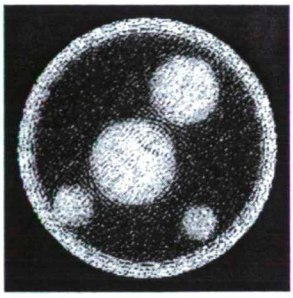


(b) 5% Gaussian noise

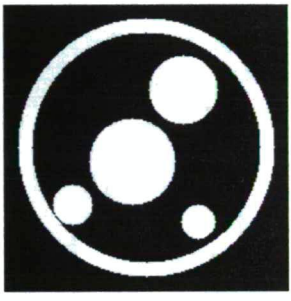
Figure 11: Relative mean error obtained from varying the number of detector elements.



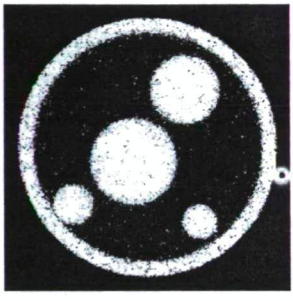
(a) 101 detector elements, no noise



(b) 101 detector elements, 5% Gaussian noise



(c) 281 detector elements, no noise



(d) 281 detector elements, 5% Gaussian noise

Figure 12: Images reconstructed from projections using a different number of detector elements ($K = 32$).

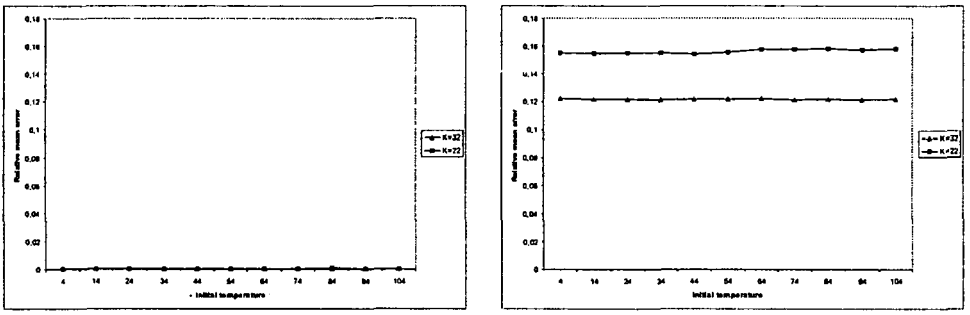
5.2 The parameters of the SA optimization algorithm

The SA algorithm we implemented has several parameters. We will investigate here what happens when we vary the initial temperature and stopping criteria.

5.2.1 Initial temperature

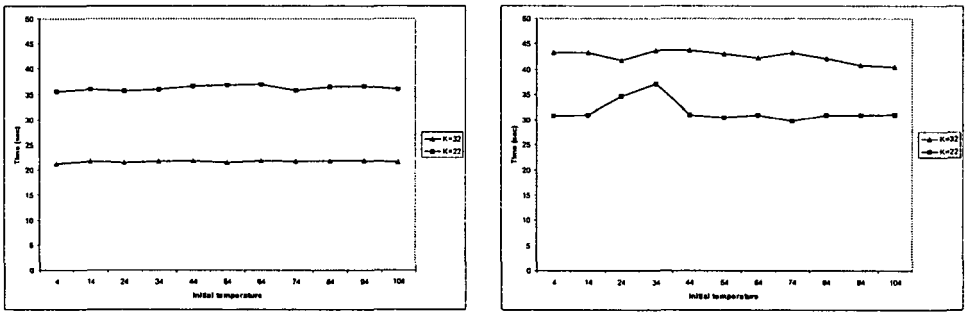
As is well known, a higher temperature in the SA algorithm may make the optimisation process go in the 'wrong' direction (thus we accept changes with a higher probability when the objective function C increases, so $C(\mathbf{x}') > C(\mathbf{x})$) . If the temperature is very low then changes in the 'wrong' direction have small probabilities. When the initial temperature is high the algorithm in the first iterations can make 'wrong' changes with a higher probability. There is then a chance of making "big jumps", thus it can get near the global minimum. However, an excessively high initial temperature can cause unnecessary iterations during the execution of the algorithm. Thus the parameter setting must be chosen with care.

In our program the initial temperature was varied from 4 to 104 (see Fig. 13).



(a) No noise (b) 5% Gaussian noise

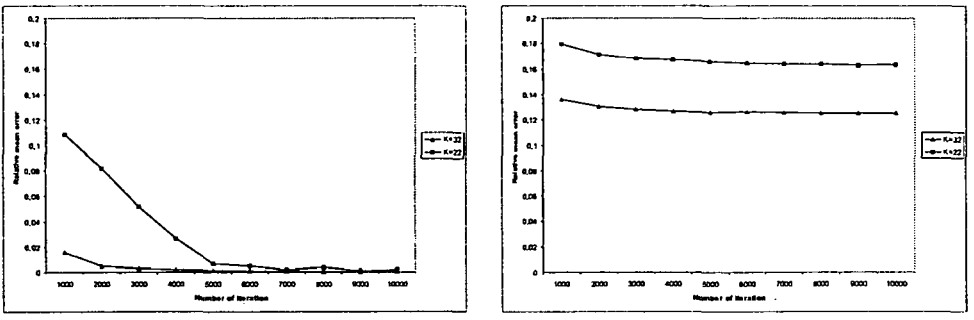
Figure 13: Relative mean error as a function of the initial temperature.



(a) No noise (b) 5% Gaussian noise

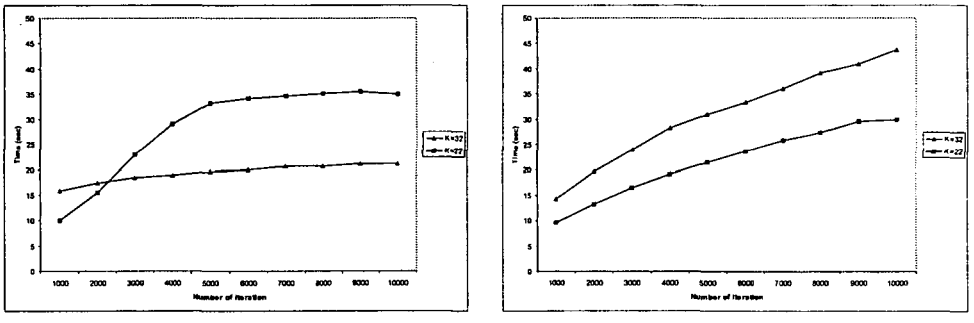
Figure 14: Execution time as a function of the initial temperature.

At the base of the curves we can see (see Fig. 13) that the influence of the initial temperature on the relative mean error is very limited. This is because these temperatures provide enough freedom for the SA method to find the global minimum. The curves of the execution times (see Fig. 14) do not show any major differences with a higher initial temperature. It is noticeable, however, that when there was no noise added, the algorithm when $K = 22$ case took more time to reconstruct the phantom image than when $K = 32$ because the fewer number of equation meant a less determined system of equations with, perhaps, more local minima. The results show the opposite case when 5% Gaussian noise was added to the projections. The explanation might be that when noise was added, the $K = 32$ case had more equations and more local minima than the $K = 22$ case.



(a) No noise (b) 5% Gaussian noise

Figure 15: Relative mean error as a function of the number of iterations in the stopping criteria ($R_{\text{thr}} = N_{\text{iter}} - 10$ and $N_{\text{var}} = N_{\text{iter}}/2$).



(a) No noise (b) 5% Gaussian noise

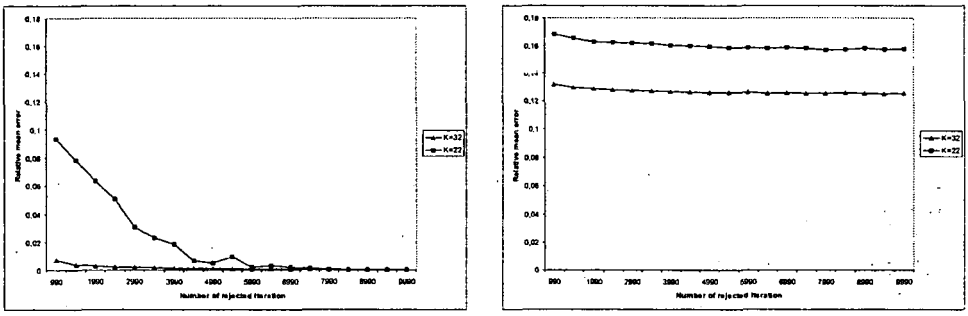
Figure 16: Execution time as a function of number of iterations in the stopping criteria ($R_{\text{thr}} = N_{\text{iter}} - 10$ and $N_{\text{var}} = N_{\text{iter}}/2$).

5.2.2 Stopping criteria

The stopping criteria in our SA algorithm is determined by counting the rejected iterations within the last N_{iter} number of iterations. If this number is greater than a given threshold (R_{thr}) then we say that the *efficiency* is low, because too few changes were accepted.

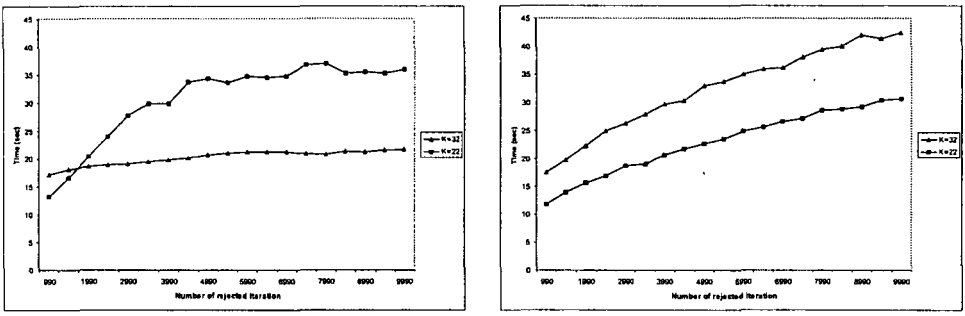
We investigated the effects of varying these two parameters separately. First, we changed N_{iter} and kept the threshold number R_{thr} high (see Figs. 15 and 16). The estimate for the variance of the objective function was calculated from the last $N_{\text{var}} = N_{\text{iter}}/2$ number of accepted changes. Second, we fixed the number of iterations and increased the efficiency (i.e. R_{thr}, the number of rejected iterations). The results are displayed in Figs. 17 and 18.

The relative mean error showed a downward trend in both experiments owing



(a) No noise (b) 5% Gaussian noise

Figure 17: Relative mean error as a function of the number of rejected iterations in the stopping criteria ($N_{\text{iter}} = 10000$ and $N_{\text{var}} = 5000$).



(a) No noise (b) 5% Gaussian noise

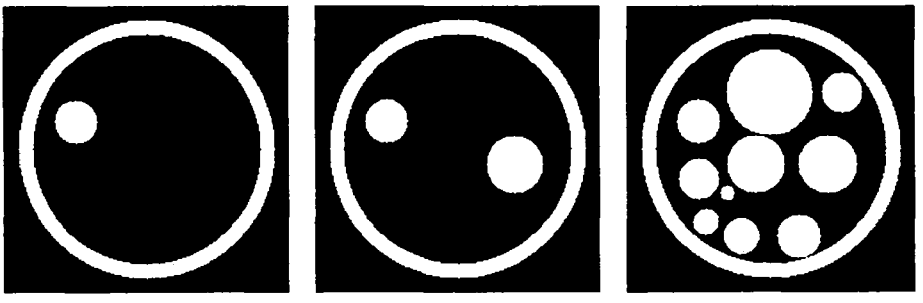
Figure 18: Execution time as a function of the number of rejected iterations in the stopping criteria ($N_{\text{iter}} = 10000$ and $N_{\text{var}} = 5000$).

to the behaviour of the algorithm used. The execution time, in contrast, showed a rising trend. From these findings we may conclude that a greater efficiency produces better results, but at a price.

5.3 Complexity

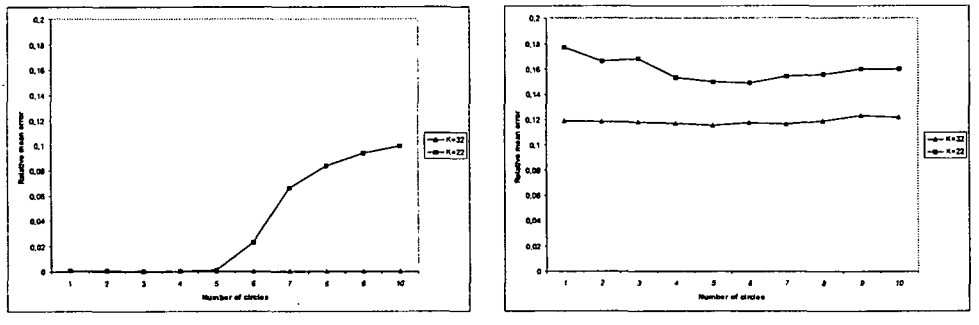
Here we generated 10 different software phantoms. These phantoms had 1, 2,..., 10 small circles inside a big ring of a given size (see Fig. 19). The experiment was repeated 100 times for each software phantom. The results of these tests are displayed in Fig. 20. The curves here clearly show that the situation is quite different for the noise-free and noisy projections. If the projections are noiseless and $K = 22$, more complex images can be reconstructed but these will have a higher error. It seems that when $K = 32$ this many sources gives a sufficient

number of equations to enable us to reconstruct complex images like the ones shown. When Gaussian noise was afterwards added to the projections the quality of the reconstruction did not change markedly. The latter was anticipated because the objective function then has more local minima than it does for the noiseless case.



(a) Ring with 1 circle. (b) Ring with 2 circles. (c) Ring with 10 circles.

Figure 19: Software phantom images made for testing the complexity.



(a) No noise (b) 5% Gaussian noise

Figure 20: Relative mean error as a function of the number of circles (complexity of the image).

5.4 The regularization parameter

In the previous cases the regularization parameter γ in (4) was assigned a value of 145. Varying the value of γ from 0 to 145 the second term becomes more important. In the experiments we used a special kind of function for $\Phi(x)$, namely

$$\Phi(x) = \Phi_{\mathrm{poz}}(x) = \sum_{j=0}^{nm-1} poz(f_j - f_j^{(0)}), \tag{7}$$

where *poz* denotes the positive part of y. Formally,

$$poz(y) = \begin{cases} y, \text{ if } y > 0 \\ 0, \text{ otherwise,} \end{cases} \tag{8}$$

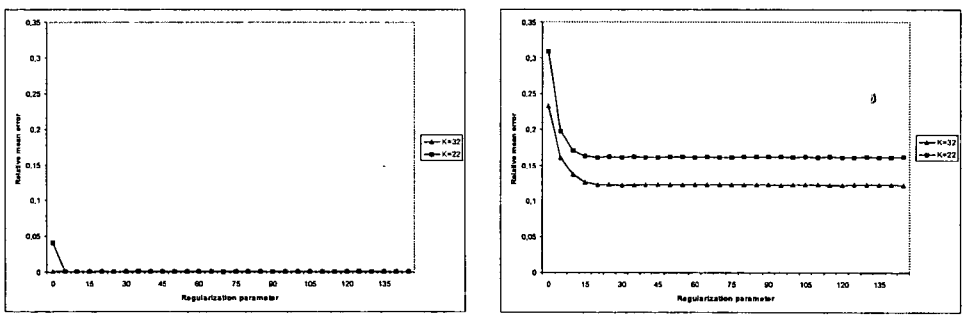
and $f_j^{(0)}$ is a so-called *prototype function*. For the phantom shown in Fig. 21(a), the prototype function $f^{(0)}$ was the mask in Fig. 21(b). The regularization parameter and penalty term bias the algorithm according to the mask function. Optimizing the objective function we get a result like this, which will lie inside the given mask.

We carried out experiments where we varied the regularization parameter γ in Φ_{poz} (see Figs. 22 and 23). We got a qualitative improvement in the noisy case using the Φ_{poz} penalty term and increasing the value of the regularization parameter (see Fig. 22(b)). We also obtained good results when we had $K = 22$ and no noise was added to the projections (see Fig. 22(a)). This is because a pixel did not change outside the $f^{(0)}$ mask, and the $\Phi_{\text{poz}}(x)$ regularization tag penalised the objective function when it did so.

(a) Software phantom.

(b) Penalty function for the software phantom.

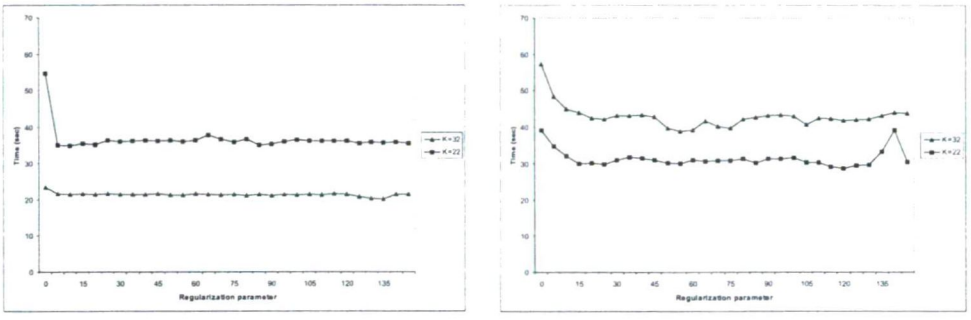
Figure 21: Images of the software phantom and the penalty function.



(a) No noise

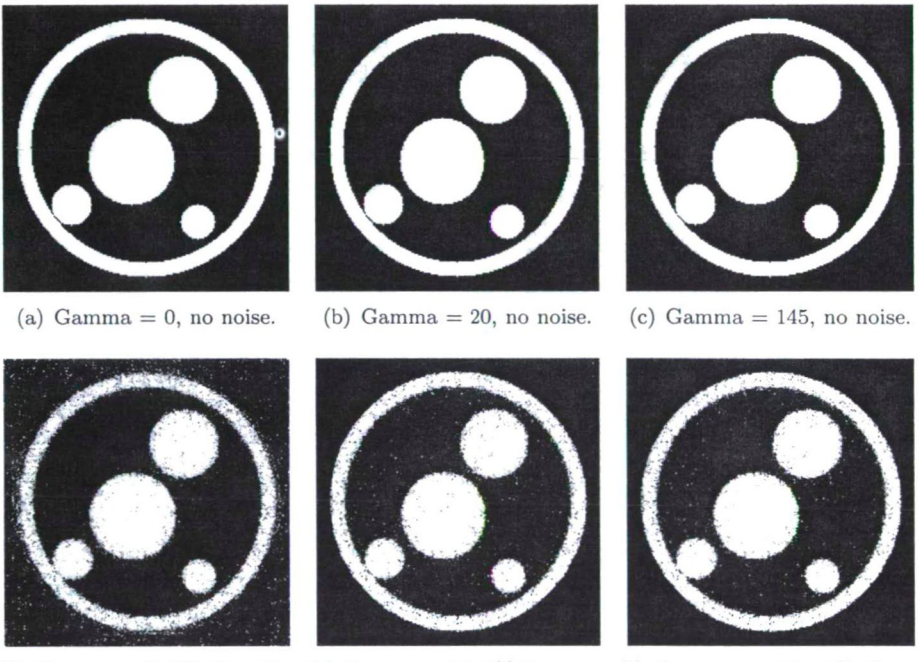
(b) 5% Gaussian noise

Figure 22: Relative mean error as a function of the regularization parameter (Φ_{poz}).



(a) No noise (b) 5% Gaussian noise

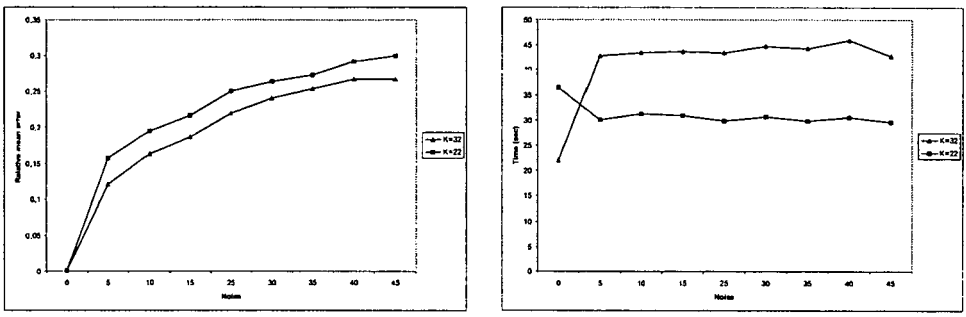
Figure 23: Execution time as a function of the regularization parameter (Φ_{poz}).



(a) Gamma = 0, no noise. (b) Gamma = 20, no noise. (c) Gamma = 145, no noise.

(d) Gamma = 0, 5% Gaussian noise. (e) Gamma = 20, 5% Gaussian noise. (f) Gamma = 145, 5% Gaussian noise.

Figure 24: Images obtained when varying the Φ_{poz} regularization parameter ($K = 32$).



(a) Relative mean error

(b) Time (sec)

Figure 25: The effect of changing the noise ratio.

5.5 Noise

The noise ratio η in our experiments was varied from 0% to 45%. Once again we focused on the relative mean error (see Fig. 25(a)) and the time (see Fig. 25(b)). Increasing the amount of noise in the projections we got worse results, as expected (see Fig. 25(a)). The algorithm was found to halt in the $K = 22$ case when we had fewer equations and more noise. The reason for this is the number of equations needed. If we have fewer equations the system is less determined in the noiseless case. In the $K = 32$ case when we add noise the number of local minima seems to be more than that for the $K = 22$ case.

6 Discussion and Conclusions

Now we will summarise the results of the previous sections. First, we studied the effects of varying the parameters of the fan-beam projections and the SA reconstruction algorithm with binary-valued matrices. With binary-valued phantoms we carried out experiments to study the effects of varying parameters on the quality of the reconstructed objects. In each test only one parameter was varied. The tests were repeated 100 times to get a better approximation for the relative mean error and for the execution time.

From our findings it is apparent that, when using the SA method, there is no major difference between reconstructing an image from a fan-beam and reconstructing one from parallel-beam projections. We saw this when we varied the distance between the sources and the origin in the experiments (see Fig. 6).

Changing the start angle when the source number was 4 yielded better results only in certain special source positions (see Fig. 7). When the number of sources (see Fig. 9) and number of detector elements (see Fig. 11) were varied in the experiments, we observed big variations in the quality of the reconstruction.

Upon increasing the initial temperature we did not, as expected, notice any

big difference between the results (see Fig. 13). Varying the stopping criteria we found that good reconstruction results could be obtained but they required longer execution times (see Figs. 15, 17, 16 and 18). The main task here, however, was to determine the stopping criteria needed to produce good reconstruction images with an acceptable execution time. We found such criteria for our particular software phantom and we expect that there should be similar criteria with other phantoms.

The results of the complexity test in Section 5.3 show that, when K was changed from 22 to 32, there was no great change in the relative mean error when noise was present (see Fig. 20(b)). In this test we found we needed a feasible number of equations (number of sources or number of detectors) when no noise was added to the projections (see Fig. 20(a)). This number was found to depend on the geometrical complexity of the phantom.

In Section 5.4 we noticed that, beyond a certain point, varying the regularization parameter did not yield better results (see Fig 22). This value was lower for the noiseless case than for the noisy case and it was found to depend on the number of equations used in the reconstruction process.

The simulated experiment in Section 5.5 with noise shows that when we have more information and noisy projections, it usually requires more time to reconstruct the object from these projections (see Fig. 25). This is because the equation system is less determined in the noisy case.

Overall, the study revealed that the number-of-equations parameter is strongly related to other parameters as well like the stopping criteria, regularization parameter and complexity of the phantom. We obtained similar results with different parameter settings and found the execution time was also an important factor in the reconstruction process. A rehearsal of the experiments will, of course, be necessary for reconstructing real objects with well-defined, measurable parameters.

Acknowledgments

This work was supported by the NSF grant DMS0306215 (Aspects of Discrete Tomography). We would also like to thank David P. Curley for checking the linguistic asspects of this paper.

References

[1] G.T. Herman, Image Reconstruction from Projections, *Academic Press*, New York, (1980).

[2] G.T. Herman and A. Kuba, Discrete Tomography. Foundations, Algorithms, and Applications, *Birkhauser*, Boston, (1999).

[3] R.A. Brualdi, *Matrices of zeros and ones with fixed row and column sum vectors*, Linear Algebra Appl. 33 (1980) 159-231.

[4] A. Kuba, G.T. Herman, *A historical introduction*, in [2], (1999).

[5] A. C. Kak and M. Slaney, *Principles of Computerized Tomographic Imaging*, New York, NY: IEEE Press, Inc., (1988).

[6] P. Grangeat, *Mathematical framework of cone beam 3D reconstruction via the first derivative of the Radon transform*, Mathematical Methods in Tomography, Lecture notes in Mathematics, eds. G.T. Herman, A.K. Louis, and F. Natterer, 1497 (1991) 66–97.

[7] N. Robert, F. Peyrin, M. J. Yaffe, *Binary vascular reconstruction from a limited number of cone beam projections*, Med. Phys. 21, (1994) 1839–1851.

[8] B. Chalmond, F. Coldefy, B. Lavayssiere, *Tomographic reconstruction from non-calibrated noisy projections in non-destructive evaluation*, Inverse Problems 15, (1999) 399–411.

[9] A. Kuba, L. Ruskó, L. Rodek, Z. Kiss: *Preliminary results of Discrete Tomography in Neutron Imaging*, in press *IEEE Trans. Nucl. Sci.* (2005).

[10] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller, *Equation of state calculation by fast computing machines*, J. Chem. Phys. 21 (1953) 1087–1092.

[11] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, *Optimization by simulated annealing*, Science 220 (1983) 671–680.

Acta Cybernetica 17 (2005) 385–399.

Classifier Combination Schemes in Speech Impediment Therapy Systems

Dénes Paczolay*, László Felföldi*, and András Kocsor*

Abstract

In the therapy of the hearing impaired one of the key problems is how to deal with the lack of proper auditive feedback which impedes the development of intelligible speech. The effectiveness of the therapy relies heavily on accurate phoneme recognition [1, 4, 17]. Because of the environmental difficulties, simple recognition algorithms may have a weak classification performance, so various techniques such as normalization and classifier combination are applied to increase the recognition accuracy. This paper examines Vocal Tract Length Normalization techniques [5, 13] focusing mainly on the real-time parameter estimation [12], and the majority of classifier combination schemes, including the traditional (*Prod*, *Sum*, *Min*, *Max*) [7], basic linear (*simple*, *weighted*, *AHP-based* [6] *averaging*), and some special linear (*Bagging*, *Boosting*) combinations. Based on the results we conclude that hybrid combinations can improve the effectiveness of the real-time normalization methods.

1 Introduction

In the therapy of the hearing impaired one of the central problems is how to deal with the lack of proper auditive feedback that hinders the development of intelligible speech. Our Phonological Awareness Teaching System, the "Speech- Master" package, seeks to apply speech recognition technology to speech therapy techniques. It provides a visual phonetic feedback for replacing the insufficient auditive feedback of the hearing impaired. We designed and implemented computer-aided training software that uses an effective phoneme recognizer and provides a realtime visual feedback in the form of flickering letters on calling pictures.

Since the system should work reliably both for children and teachers of different ages, the recognizer has to be trained with the voices of users of both genders and of practically any age. The task is also special because it has to recognize isolated phones, so it cannot rely on language models. Consequently, there is a heavy burden on the acoustic classifier, and we need to apply any helpful trick that might improve the overall performance.

*Research Group on Artificial Intelligence of the Hungarian Academy of Sciences and University of Szeged. H-6720 Szeged, Aradi vértanúk tere 1., Hungary. E-mail: {pdenes, lfelfold, kocsor}@inf.u-szeged.hu,Web: http://www.inf.u-szeged.hu/speech

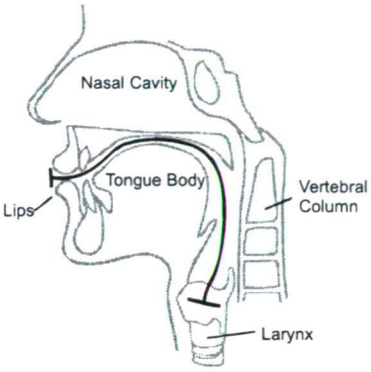


Figure 1: Vocal Tract Length

One such technique is speaker normalization, or more specifically, vocal tract length normalization (VTLN), which proves very useful when the targeted users vary greatly in age and gender. Applying off-line vocal tract length normalization algorithms [3, 5, 13, 16, 18, 19, 21] one can build recognizers that work robustly with voice samples from males, females and children. However, we were also faced with the requirement that the normalization parameters had to be estimated on-line. To solve this problem we used nonlinear regression methods of machine learning [12]. Based on the experiments on-line approximations closely approach the recognition results of the off-line methods.

Another technique is classifier combination [11, 20], which aggregates the results of many classifiers, overcoming the possible local weakness of the individual inducers, thus producing a more robust classification performance. Classifier combinations offer classifier methods for improving their recognition accuracy.

Our aim is to examine how the combination techniques affect the performance of the on-line normalization techniques, and we will show that with a careful selection of combiners it is possible to surpass the accuracy of the off-line methods.

This paper is organized as follows. The following section gives a brief description of the Vocal Tract Length Normalization, and then examines the various parameter estimation techniques. In Section 3 we offer an overview of classifier combination techniques, focusing on the traditional and linear combination schemes. The experimental section compares the performance of the various VTLN and combination methods. Lastly, we give some brief conclusions and ideas for future research.

2 Vocal Tract Length Normalization

One of the major physiological sources of inter-speaker variation is the vocal tract length of the speakers (Fig. 1). In [3] the average vocal tract length for men was reported to be 17 cm, for women it was 15 cm, and for children it was 14 cm. The formant frequency positions are inversely proportional to vocal tract length and this causes a shift of the formant center frequencies. Consequently, VTLN is

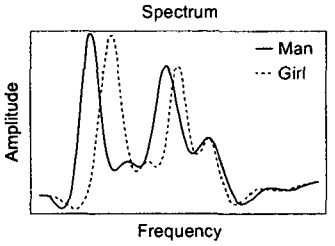


Figure 2: Vocal Tract Length dependent Frequency shifting. The graph drawn with solid and dashed line shows the spectrum of a vowel uttered by a man and a girl, respectively.

usually performed by warping the frequency scale.

Modelling the vocal tract as a uniform tube of length L, the format frequencies are proportional to 1/L. Thus the simplest approaches use a linear warp. In reality, however, the vocal tract is more complex than a uniform tube. That is why many more sophisticated warping functions have been proposed in the literature [5, 18]. Some of the commonly applied warping functions are shown in Figure 3.

Given a warping function, the normalization can be implemented either by re-sampling and interpolating the spectrum or modifying the width and center frequencies of the mel (Bark) filter bank.

2.1 VTLN parameter estimation

The linear discriminant (LD) criterion is defined using the covariance matrices of a given sample set over a speech database. Each sample is placed in a phonetic class and the samples that belong to a given speaker are extracted using the same warping parameter. The task is then to optimize these parameters for each speaker according to the LD criterion:

$$LD = \frac{|B|}{|W|}, \tag{1}$$

where B is the between-class and W is the within-class covariance matrix. The value of the LD criterion is small if the different classes are spaced out and each of them has a small scatter around the class centers. While optimizing the warping parameters of all the speakers at the same time is impractical, an iterative process (Algorithm 1) can be applied.

This optimization method, however, works off-line. So a natural question then arises. Is it possible to efficiently estimate the optimal parameters obtained by off-line algorithms using machine learning regression methods that work on-line?

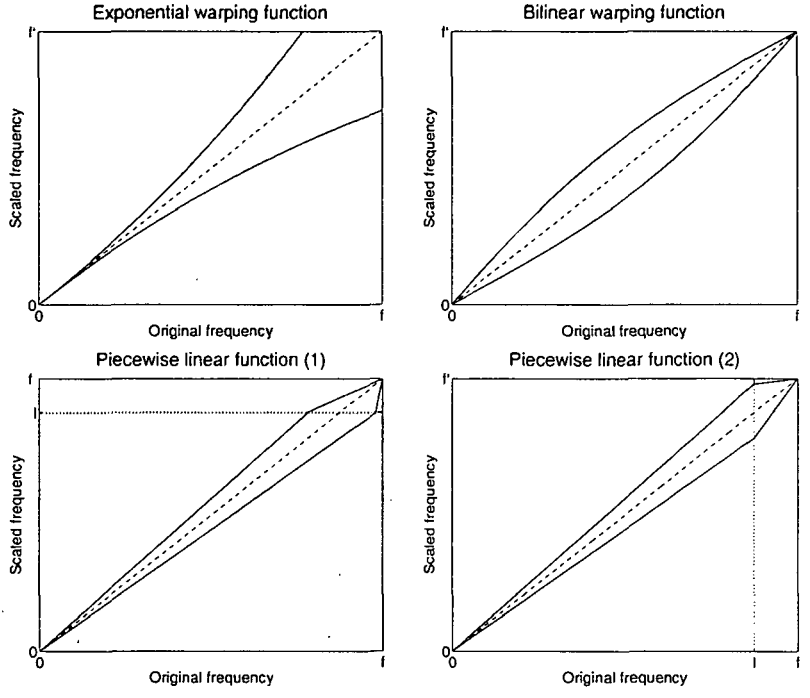


Figure 3: Examples of VTL warping functions. The figures show the mapping between the original (horizontal axis) and the warped (vertical axis) frequencies.

Algorithm 1 LD-VTLN parameter estimation

Choose an initial warping factor for each speaker and warp the samples
while the average warping factor variation is above the set threshold **do**
  for all speakers **do**
    calculate the LD criterion for each α value in a small neighborhood of the current warping parameter
    select the best warping parameter
  end for
  Update the sample set using the optimal warping factors that are obtained
end while

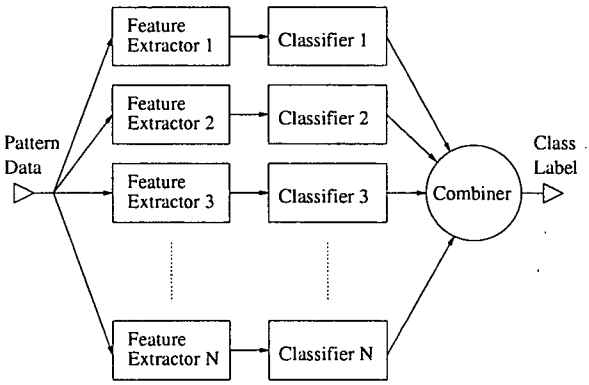


Figure 4: General parallel combination scheme

2.2 Real-time VTLN

LD-VTLN parameter estimation requires all the utterances in advance. Real-time recognition systems, however, require an instantaneous response. To have the advantages of the speaker normalization in on-line systems, machine learning methods can be applied to the estimation of the correct parameters. Out of the many possible regression techniques we chose to experiment with neural nets. Their task was to estimate the optimal LD-VTLN warping parameter for each speaker based on the actual spectral frame without warping. One of our previous papers [12] compared the performance of these kinds of on-line method with those off-line LD-VTLN parameter estimation techniques.

3 Classifier Combinations

Classifier combination (see Fig. 4) is an effective way of improving classification performance. It aggregates the results of many classifiers, overcoming the possible local weakness of the individual inducers, producing a more robust recognition. From a combination viewpoint, classifiers can be categorized into the following types:

- abstract: the classifier yields only the most probable class label
- ranking: it generates a list of class labels in order of their probability
- confidence: the probabilities for each class are known

In the following we will only deal with the confidence type.

3.1 Selecting Classifiers

To work optimally, a combination requires nearly independent classifiers. In practice there are special techniques for generating the appropriate classifier sets:

- Training using different parameter sets
- Training on different data-sets
  - Database Rotation: the dataset is divided into n parts and reassembled using $m < n$ parts (n-fold cross-validation, $m = n - 1$)
  - Bootstrapping: the elements are randomly drawn with replacement from the same data-set (Bagging),
  - Weighted bootstrapping: like Bootstrapping, but the elements are drawn according to a given distribution (Boosting),
- Inserting random noise.

3.2 Combinations

In the following let x mean a pattern, and $(\omega_1, \ldots, \omega_n)$ the set of possible class labels. p_i^j will represent the output of i-th classifier for the j-th class. Furthermore, let $\mathcal{L}(x)$ denote the correct class labelling for each training sample $x \in S$, and C_i refer to a function that maps the pattern x to the class label assigned by the i-th classifier:

$$C_i(x) = \omega_k, \quad k = \underset{j}{\operatorname{argmax}}\, p_i^j(x).$$

The combined class probabilities $\hat{p}^j$ are calculated from the corresponding values of classifiers p_i^j according to combination rules described later. The class label $\hat{C}(x)$ selected by the combiner is the one with the largest probability:

$$\hat{C}(x) = \omega_k, \quad k = \underset{j}{\operatorname{argmax}}\, \hat{p}^j(x).$$

There are numerous combination rules mentioned in the literature. The traditional combination methods are listed here:

Product Rule:

$$\hat{p}^j(x) = \prod_{i=0}^{N} p_i^j(x) \tag{2}$$

Sum Rule:

$$\hat{p}^j(x) = \sum_{i=0}^{N} p_i^j(x) \tag{3}$$

Max Rule:

$$\hat{p}^j(x) = \max_{i=0}^{N} p_i^j(x) \tag{4}$$

Min Rule:

$$\hat{p}^j(x) = \min_{i=0}^{N} p_i^j(x) \tag{5}$$

Voting Rule:

$$\hat{p}^j(x) = \sum_{i=0}^{N} \delta_{\mathcal{C}_i(x),\omega_j} \tag{6}$$

Here δ_{ij} is the Kronecker symbol:

$$\delta_{ij} = \begin{cases} 1 & \text{if } i = j \\ 0 & \text{otherwise} \end{cases}$$

3.3 Linear combiners

In the case of linear combinations, the elements of the combined class probability vector $(\hat{p}^j)$ are equal to the linear combination of the output classifiers p_i^j, i.e. :

$$\hat{p}^j(x) = \sum_{i=1}^{N} w_i p_i^j(x) \tag{7}$$

Various methods can be applied to determinate the weighting factors w_i, e.g.:

- Simple Averaging
- Weighted Averaging
- AHP-based Linear Combination
- Minimal Squares Combination
- Bagging
- Boosting

In the following we will provide brief descriptions of these methods.

3.4 Averaging Techniques

In *Simple Averaging* the calculation of the weights is very simple. The elements of the vector w can be the same constant:

$$w_i = \frac{1}{N}. \tag{8}$$

This method is thus equivalent to the traditional *Sum Rule*. As a more general solution one might apply *weighted averaging* [9, 10], where the weights of the classifiers are chosen to be proportional to their performance measured on a selected data set S':

$$w_i = \frac{1}{E_i}, \tag{9}$$

where

$$E_i = \sum_{x \in S'} \mathcal{C}_i(x) \neq \mathcal{L}(x). \tag{10}$$

3.5 AHP-Based Linear Combination

The Analytic Hierarchy Process (AHP) [15] is an intuitive and efficient method for Multi-Criteria Decision-Making (MCDM). AHP-based combinations [6] apply a pairwise comparison matrix A to calculate the weighting factors. The a_{ij} element of the comparison matrix A represents the relative "importance" of the i-th classifier against the j-th one. This relative "importance" refers to the relative performance of the classifiers measured on a randomly generated data-set, while the performance is calculated as the reciprocal of the number of wrongly qualified patterns.

$$a_{ij} = \frac{\sum_{x \in S_{ij}} C_j(x) \neq \mathcal{L}(x)}{\sum_{x \in S_{ij}} C_i(x) \neq \mathcal{L}(x)} \tag{11}$$

Let us now focus on the computation of the weights w for a selected criterion. The elements of a given pairwise comparison matrix approximate the relative importance of the choices, hence

$$a_{ij} \approx \frac{w_i}{w_j}, \tag{12}$$

where the elements of the unknown vector w are the importance weighting factors. A matrix M is said to be consistent if its components satisfy the following set of equalities:

$$m_{ij} = \frac{1}{m_{ji}}, \tag{13}$$

and

$$m_{ij} = m_{ik} m_{kj} \quad \forall\, i, j, k. \tag{14}$$

If A is not consistent, it is not possible to find a vector w that satisfies the equation

$$a_{ij} = \frac{w_i}{w_j}. \tag{15}$$

To compute the weighting factors, one way might be to calculate the eigenvalues of the comparison matrix A and get the eigenvector corresponding to the largest eigenvalue. A way of measuring the consistency of the matrix A is by defining the *consistency index* (CI) as the negative average of the remaining eigenvalues:

$$CI = \frac{\sum_{\lambda < \lambda_{max}} \lambda}{n-1} = \frac{\lambda_{max} - n}{n-1} \tag{16}$$

If all the performance errors are measured on the same test data set, i.e. $S_{ij} = S$ for all possible i and j pairs, the comparison matrix A will be consistent, and the elements of the eigenvector whose corresponding eigenvalue will be N, that is

$$w_i = \frac{1}{E_i} \tag{17}$$

are the same as those as generated by *weighted averaging*. However, this method allows us to make pairwise comparisons of different inducers applied on different (e.g. randomly generated) test sets, taking advantage of the stabilizing effect of AHP. This leads to more a robust classification performance, especially in noisy environments.

3.6 Least Squares Combination

With the Least Squares Combination method the weights are selected to minimize the sum of squares between the combined $\hat{p}^j(x)$ and the desired target $t^j(x)$ class probabilities:

$$\sum_{x \in S} \sum_{j} [\hat{p}^j(x) - t^j(x)]^2 \tag{18}$$

The $t^j(x)$ target functions can be the same as those for artificial neural networks, namely:

$$t^j(x) = \begin{cases} 1 & \text{if } \mathcal{L}(x) = \omega_j \\ 0 & \text{otherwise} \end{cases} \tag{19}$$

3.7 Bagging

The Bagging (Bootstrap aggregating) algorithm [2] makes classifiers generated by different bootstrap samples (replicates) vote for the class label. A bootstrap sample is generated by uniformly sampling m instances from the training set with replacement. T bootstrap samples $B_1, B_2, ..., B_T$ are generated and a classifier C_i is built from each bootstrap sample B_i. A final classifier $\hat{C}$ is built from $C_1, C_2, ..., C_T$ whose output is the class predicted most often by its sub-classifiers (majority voting).

Algorithm 2 Bagging algorithm

Require: Training Set S, Inducer $\mathcal{I}$
Ensure: Combined classifier $\hat{C}$
  for $i = 1 \ldots T$ **do**
    S' = bootstrap sample from S
    $C_i = \mathcal{I}(S')$
  end for
  $\hat{C}(\mathbf{x}) = \underset{j}{\text{argmax}} \sum_{i:C_i(\mathbf{x})=\omega_j} 1$

3.8 Boosting

Boosting[8] was introduced by Shapire (1990) as a method for boosting the performance of a weak learning algorithm. Here we will focus on AdaBoost, sometimes

called "AdaBoost.M1". Like Bagging, the AdaBoost algorithm generates a set of classifiers and makes a decision based on their votes. However, in other ways, the two algorithms substantially differ. The AdaBoost algorithm generates the classifiers sequentially, while Bagging can generate them in parallel. AdaBoost also changes the weights of the training instances provided as input for each inducer, based on classifiers that were previously built. The final decision is made using a weighted voting scheme for each classifier, and the weights will depend on the performance of the training set used to build it.

Algorithm 3 Boosting algorithm

Require: Training Set S of size m, Inducer $\mathcal{I}$
Ensure: Combined classifier $\hat{C}$
  $S' = S$ with weights assigned to be $1/m$
  for $i = 1 \ldots T$ **do**
    $S' =$ bootstrap sample from S
    $C_i = \mathcal{I}(S')$
    $\epsilon_i = \sum_{\mathbf{x} \in S' : C_i(\mathbf{x}) \neq \mathcal{L}(\mathbf{x})}$ weight of $\mathbf{x}$
    if $\epsilon_i > 1/2$ **then** Exit
    $\beta_i = \frac{\epsilon_i}{(1-\epsilon_i)}$
    for all $\mathbf{x_j} \in S'$ such that $C_i(\mathbf{x}) = \mathcal{L}(\mathbf{x})$ **do**
      weight of $\mathbf{x}$ = weight of $\mathbf{x} \cdot \beta_i$
    end for
    normalize the weights of the instances
  end for
  $\hat{C}(\mathbf{x}) = \underset{j}{\operatorname{argmax}} \sum_{i:C_i(\mathbf{x})=\omega_j} \log \frac{1}{\beta_i}$

The AdaBoost algorithm requires a weak learning algorithm whose error is bounded by a constant strictly less than $1/2$. In the case of multi-class classification this condition might be difficult to guarantee. Some implementations of AdaBoost make use of boosting by re-sampling because the inducers employed are unable to support weighted instances. Using appropriate classifiers one can try re-weighting, which might work better in practice.

3.9 Multi-Model classifier

Multi Model Classifiers is a special classifier combination, where the combined class probabilities are given by:

$$\hat{p}^j(x) = \sum_{i=1}^{N} w_i(x) p_i^j(x) \tag{20}$$

Here the weight vector w depends on the current pattern. In addition only one component of w can be non-zero, so this weighting selects a classifier whose output appears as the output of the given combination, i.e.:

$$w_i(x) = \begin{cases} 1 & \text{if } i = \sigma(x) \\ 0 & \text{otherwise} \end{cases}, \tag{21}$$

where σ is a special function which selects a classifier for the current pattern. To implement this function any possible machine learning method can be applied.

4 Experiments and Evaluation

Firstly we will describe the corpus, the feature extraction technique, then the classifiers and regression algorithms used in the tests.

- **Corpus:** For training and testing purposes we recorded samples from 240 speakers, namely 60 women, 60 men, 60 girls and 60 boys. The children were aged between 6 and 9. The speech signals were recorded and stored at a sampling rate of 22050 Hz in 16-bit quality. Each speaker uttered all the Hungarian vowels, one after the other, separated by a short pause. Since we decided not to discriminate their long and short versions, we only worked with 9 vowels altogether.

- **Feature Sets:** The signals were processed in 10 ms frames, the log-energies of 24 critical-bands being extracted using FFT and triangular weighting [14]. The energy of each frame was normalized separately, which meant that only the spectral shape was used for classification.

- **Classifiers:** In all the classification experiments the Artificial Neural Nets (ANNs) [1] employed here were the well-known three-layer feed-forward MLP networks trained with the back-propagation learning rule. The number of hidden neurons was equal to 16.

4.1 Warping parameter estimation

In order to test the performance of the VTLN techniques, we transformed the original features of the databases using the calculated α warping parameter for each pattern and then generated a database for each set of output data. To estimate parameter values the following methods were applied:

- **LD-VTLN:** The initial value of the warping parameter α was set to 0. For the optimization the value of α in this interval was quantized – it could take one of 15 discrete values. The iteration was stopped when the average change in the warping parameter fell below 10^{-2}.

- **RT-VTLN:** For the learning of the parameter α of the warping function a special MLP network was constructed with one output neuron and two

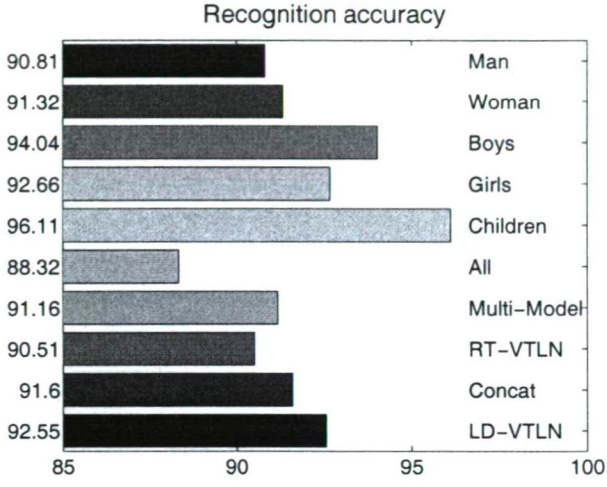


Table 1: Recognition accuracy without combination (in percent)

hidden layers with 32 and 24 neurons, respectively. Training was performed with respect to mean square error.

4.2 Tests without combination

The experiments were conducted as follows. We employed 5-fold leave-one-out cross-validation, keeping the ratio of boys, girls, men and women uniform in each case. So the train and test sets had a ratio of 4 : 1. Separating the original database named "All" according to the speakers gender and age, we obtained databases for "Men", "Women", "Boys", "Girls", and "Children". On each database we trained an ANN classifier. For the database "Multi model", we divided the train set into 3 categories: men, women, and children. On each of them we trained an ANN as classifier, and afterwards we trained an ANN with 16 hidden neurons to select the category of a given pattern. Because the parameter estimation of LD-VTLN requires all pattern data in advance, this method cannot be utilized in real recognition systems, so we treated its performance as a reference value for the other normalization techniques. To make our regression-based normalization RT-VTLN more robust, we generated a database "Concat" that, besides the warped features, contains the original features as well.

Table 1 shows the classification accuracies measured on the chosen databases. It was clear that the performance on the separated categories was significantly better than that of the original, the 'Multi Model' method being based on this experiment. Of the VTLN techniques we applied, LD-VTLN performed the best, improving the accuracy by 36 %. But being off-line technique, it requires all data in advance to work, so this cannot be applied to real classification problems. The run-time VTLN

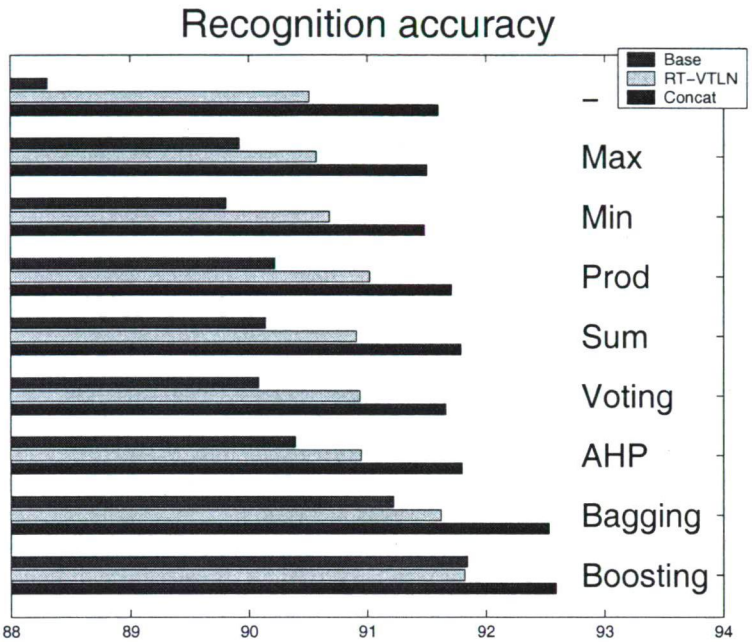


Table 2: Recognition accuracy on the databases with combination (in %). The various bars in each triplet correspond to the databases, and bar-triplets represent the applied combiner.

produced a moderate performance compared to the LD-VTLN, but this difference could be halved using not only the warped but the original features ("Concat").

4.3 Tests with combination

For the combiners "Max", "Min", "Prod", "Sum", "Voting", and "AHP" we exploited the 8-fold rotation of the database to generate 8 data-sets as training sets for the classifiers. After training ANN-s on the corresponding sets we utilized a given set of combiners on them, and measured their performances on a common test data-set. "Bagging" and "Boosting" generated their own sequence of training data-sets; in these cases we ran the methods on the original database directly, setting the max iteration number to 50.

The experimental results are shown in Table 2. For each combiner the recognition accuracy was measured on 3 different databases: "All", "RT-VTLN", and "Concat". The effect of the classifier combination depends on the database complexity. With the "All" database, each of the combiners has a better performance than that for the original classification. On the warped databases ("RT-VTLN" and "Concat") the traditional combinations have less influence. Bagging and Boosting

gave the best scores due to the large number of classifiers used.

Comparing the above results with the reference figure of LD-VTLN (92.55 %), we may conclude that with a properly selected combination scheme the regression based real-time VTLN method (Boosting on Concat, 92.67%) can outperform the results of the off-line method LD-VTLN.

5 Conclusions

In this paper we examined the effect of Vocal Tract Length Normalization techniques with classifier combination methods on classification performance. Based on the test results here we can say that the off-line method LD-VTLN can decrease the recognition error of the base classifier by 36 %. Using our regression-based on-line estimation of VTLN parameter (RT-VTLN), the difference in performance between the base classifier and LD-VTLN method can be halved. Classifier combinations further improve the performance, achieving nearly the same performance as the off-line normalization version, while applying Bagging and Boosting may produce classifiers with better performances than those for LD-VTLN. If they do, we can produce a more robust speech recognition system for speech impediment therapy.

References

[1] C. M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[2] Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

[3] T. Claes, I. Dologlou, L. Bosch, and D. Compernolle. A novel feature transformation for vocal tract length normalization in automatic speech recognition. *EEE Trans. on Speech and Audio Processing*, 6:549–557, 1998.

[4] R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification.* John Wiley and Son, New York, 2001.

[5] E. Eide and H. Gish. A parametric approach to vocal tract length normalization. In *ICASSP*, pages 1039–1042, Munich, 1997.

[6] L. Felföldi and A. Kocsor. Ahp-based classifier combination. In *The 4th International Workshop on Pattern Recognition in Information Systems (PRIS-2004)*, Porto, 2004.

[7] L. Felföldi, A. Kocsor, and L. Tóth. Classifier combination in speech recognition. *Periodica Polytechnica*, 47(1-2):125–140, 2003.

[8] Freund. Boosting a weak learning algorithm by majority. In *Proceedings of the Workshop on Computational Learning Theory (COLT 1990).* Morgan Kaufmann Publishers, 1990.

[9] G. Fumera and F. Roli. Performance analysis and comparison of linear combiners for classifier fusion. In *IAPR Int. Workshop on Statistical Pattern Recognition (SPR 2002)*, 2002.

[10] G. Fumera and F. Roli. Linear combiners for classifier fusion: Some theoretical and experimental results. In *4th Int. Workshop on Multiple Classifier Systems (MCS 2003)*, Guildford, UK, January 2003. Springer-Verlag, LNCS.

[11] Anil K. Jain, Robert P. W. Duin, and Jianchang Mao. Statistical pattern recognition: A review. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):4–37, 2000.

[12] D. Paczolay, A. Kocsor, and L. Tóth. Real-time vocal tract length normalization in a phonolgical awareness teaching system. In *Text Speech and Dialogue*, volume 2807, pages 4–37, Czech Republic, 2003. Springer.

[13] P. Pitz, S. Molau, R. Schlter, and H. Ney. Vocal tract normalization equals linear transformation in cepstral space. In *EUROSPEECH*, volume 4, pages 2653–2656, Denmark, 2001.

[14] L. R. Rabiner and B. H. Juang. *Fundamentals of Speech Recognition.* Englewood Cliffs, NJ, Prentice Hall, 1993.

[15] T. L. Saaty. *The Analytic Hierarchy Process.* McGraw-Hill, New York, 1980.

[16] L. F. Uebel and P. C. Woodland. An investigation into vocal tract length normalisation. In *EUROSPEECH*, volume 6, pages 2527–2530, Hungary, 1999.

[17] V. N. Vapnik. *Statistical Learning Theory.* John Wiley and Son, 1998.

[18] S. Wegmann, D. McAllaster, J. Orloff, and B. Peskin. Speaker normalization on coversational telephone speech. In *ICASSP*, volume 1, pages 339–341, Atlanta, 1996.

[19] M. Westphal, T. Schultz, and A. Waibel. Linear discriminant - a new criterion for speaker normalization. In *ICSLP*, number 755, Sydney, 1998.

[20] L. Xu, A. Krzyzak, and C.Y. Suen. Method of combining multiple classifiers and their application to handwritten numeral recognition. *IEEE Trans. on SMC*, 22(3):418–435, 1992.

[21] P. Zhan and M. Westphal. Speaker normalization based on frequency warping. In *ICASSP*, volume 1, pages 1039–1042, Munich, 1997.

Acta Cybernetica **17** (2005) 401–417.

Combining metric and topological navigation of simulated robots

Richárd Szabó*

Abstract

Mobile robotics and robot navigation is a growing area of scientific research. Robot simulators are useful designing and analizing tools of this domain.

Webots [1] is a well-known representant of these programs, a three-dimensional mobile robot simulator. Various guidance principles can be developed in C/C++ or Java programming language with the use of Webots controller programs.

In this paper a short overview is given about the problems arising in the process of the navigation, and a short taxonomy is presented about the possible problem solving methods [2]. A brief introduction to the probabilistic navigation techniques concerning Kalman filter and expectation maximization is included with a special focus on occupancy grid.

Formerly I presented a metric navigation method based on occupancy grid working in the Webots simulation environment [3]. As a continuation of that research I created an enhancement of the former processes, a hybrid metric-topological navigation mechanism. A topologic layer is introduced in the environment exploration phase replacing the older value iteration [4]. The implementation of a topologic graph of the explorable places using the metric map enables the robot to navigate in a more efficient manner. A comparison of the pure metric and the new hybrid methods is also given.

Keywords: robot simulation, probabilistic mapping, occupancy grid, metric/topological navigation

1 Introduction

Mobile robotics can be the main propellant power of the development in the 21st century. If robots will be able to take up humans' not liked, monotone, everyday tasks then the world will change more revolutionarily than with the rise of the internet.

*Department of Software Technology and Methodology, Faculty of Informatics, Eötvös Loránd University, 1117, Pázmány P. s. 1/D. Budapest, Hungary. Department of History and Philosophy of Science, Eötvös Loránd University, 1117, Pázmány P. s. 1. Budapest, Hungary.

The frequently heard answer to the question 'When will we reach this new era?' is in ten years. The underestimation of the problem is driven by several factors, high hopes of artificial intelligence is one of them. After all, an important component of a future robot servant is a navigation module that can map and travel the environment without human aid.

In this paper I present a method for building topological navigation graph on the top of an occupancy grid in the Webots simulator. First of all I list some basic problems of navigation. Then I outline a taxonomy of probabilistic mapping methods. After that I show in brief the creation of an occupancy grid and the environment exploration with value iteration. Finally I focus on the necessary steps of the composition of the graph and the environment exploration utilizing the evolved graph.

This project is part of my Ph.D. research with the main aim of the investigation of mobile robot navigation. The primary tool for the experiments is the Webots mobile robot simulator. After I was the runner up of the 1st Artificial Life Creators Contest organized by Cyberbotics Ltd. in 1999, I won the second contest in 2000 and obtained the simulator license as the first prize. Details of the competitions are discussed in [5].

2 The problem of navigation

During navigation the robot tries to determine the position of important objects and itself. For this reason usually an internal map is handled. The task is twofold: the robot has to navigate among objects using the actual map while it has to refine the map with new measurements of the environment at the same time. The problem is called *simultaneous localization and mapping* (SLAM) in the literature [6]. In addition to this chicken-egg task other problems also arise.

The noise accompanying movement commands and sensor measurements worsen the quality of navigation. If the noise is indepent of the position of the robot and the time of the action then more gathered information leads to convergence after a while. However in most cases noise has not got this sympathic property, for instance measurement errors accumulate.

Another problem stems from multidimensionality of the environment: a detailed two-dimensional map of a room contains many thousand elements, that increases computational costs.

Problem of data association – which is probably the most important one – raises the issue how it is possible to associate map elements of various viewpoint and various time. For instance when a robot arrives to its starting point after the exploration of a circular gallery how can it realize that the place was already visited.

Fourth navigation problem results from the changes of the environment. Static maps generated during the exploration of experimental terrains cannot hold on in

real world where people are walking, doors are opening and closing, or when the robot finds itself in a rearranged flat.

Beyond the others a well-performing navigation procedure has to work in real time and has to be trustworthy. The final goal is generality, that is to say the robot has to be capable of navigating in universal environment.

A good example of the state of the research and technology is the DARPA Grand Challenge in the Mojave desert in March 2004 [7]. The best robot could accomplish only 5 % of the targeted task (Figure 1). This fact justifies that autonomous navigation is a really hard task.

Figure 1: An unlucky contestant

2.1 Probabilistic mapping

Navigation methods spred seriously in the nineties generally stands on probabilistic base. One of the main reasons behind this fact can be that uncertainities caused by noise are hard to handle with a monotonic map where later changes do not exist. Common property of these methods is the usage of Bayes-theorem [8] generating a maximum-likelihood map, that is to say they create the most probable map according to the actual data.

2.1.1 Kalman filter

The Kalman filter is an efficient resolution of a linear difference-equation well-known in other domains as well. The filter tries to find the noisy solution recursively on a discrete timescale [9].

First half of the problem is described below.

$$x_k = Ax_{k-1} + Bu_{k-1} + w_{k-1}$$

Here x_k is the vector describing the state of the environment at moment k, in case of navigation, it is the position of the robot and the neighbouring objects.

x_k mainly depends on the previous state (x_{k-1}), secondly the commanding actions (u_{k-1}), while there is a Gaussain noise as well (w_{k-1}).

The robot does not sense the state of the environment directly, rather through its sensors. It is modelled by the second equation.

$$s_k = Hx_k + v_k$$

Here s_k means the perceived environment, also biased with a Gaussian noise (v_k) .

The Kalman filter solves the above difference-equation with the knowledge of A, B, H matrices and the parameters of the noise distributions.

Positive property of the filter is the iterativity, that is to say previous calculations can be used in the actual estimation. Drawback of the method is the lack of data assocation.

2.1.2 Expectation maximization

This method is also a general tool for finding unknown parameters of a system with sampling [2, 10]. Expectation maximization means the alternative repetition of two steps until convergence. Firstly – during navigation – the expected value of the robot position has to be calculated using the actual map and the senses.

$$Q(m|m_k) = E_{m_k}[p(x, s|m_k)|s]$$

In the second step the actual position and the perception determine the alteration of the map, namely most probably map is searched in the space of maps.

$$m_{k+1} = argmax_m Q(m|m_k)$$

Expectation maximization gives an efficent solution of data association contrary to Kalman filter, moreover it handles arbitrary noise distribution. A major disadvantage of the method is that it does not work iteratively. The map has to be prepared from scratch every time step, or an auxiliary method is necessary. Hence only offline learning is possible because of the immense calculation costs.

2.1.3 Occupancy grid

This general structure manages a tesselation of the plane or space in cells. Each cell of the occupancy grid contains a probability value which is an estimation that the represented position is occupied by some object. Advantages of this method are that it is simple to implement and the iterative work. Among drawbacks worth mentioning the precondition of independent noise and the difficulty to navigate with.

3 Previous work

My former goal was to create a metric navigation module for a modified Khepera robot in the Webots simulation environment, that is to say I focus on metric spatial properties of objects like distances, and coordinates. The developed robot has to build a cognitive map — "a view from above" — of a square-shaped room in the size of a few square meters while it visits every reachable location [3]. Figure 2, Figure 3, Figure 4, and Figure 5 show some typical experimental area.

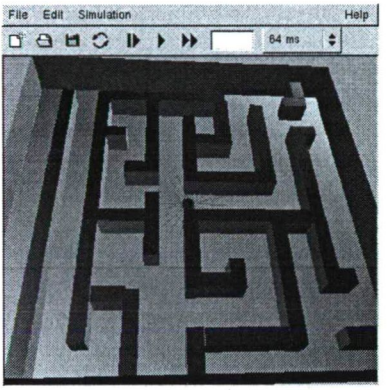


Figure 2: A maze in Webots

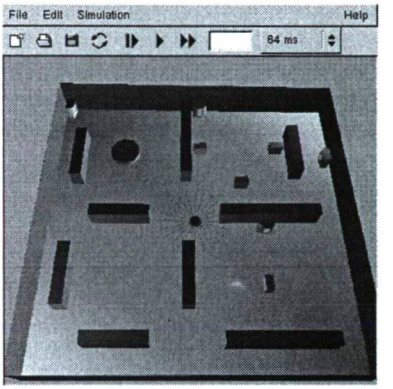


Figure 3: An office-like room

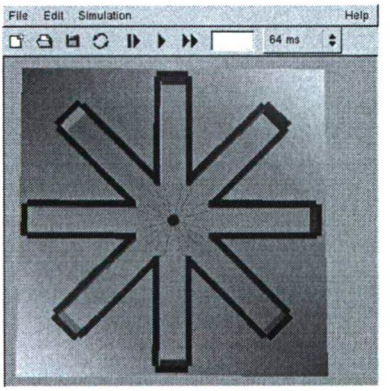


Figure 4: Radial maze

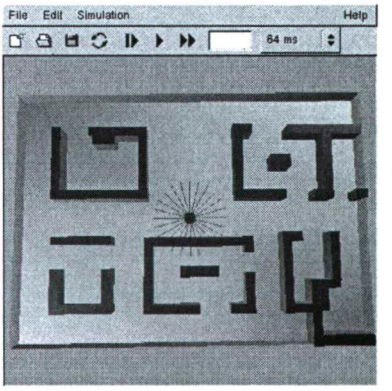


Figure 5: AAAI contest maze

The adopted method of metric navigation is based on the occupancy grid model pioneered by Moravec and Elfes [11, 12]. The important steps of the map building, in accordance with Thrun's work [4], are the following:

- sensor interpretation
- integration over time
- pose estimation
- global grid building
- exploration with value iteration

During sensor interpretation sonar scalar values are converted to occupancy values around the robot, which means that $p(occ_{x,y}|s)$ probabilities are determined for every s sonar measurement of cell (x, y).

The integration phase summarizes different measurements at different time. Using Bayes-theorem and the assumption that sensations are independent of the time of their collections, complex occupancy probabilities can be calculated with the following equation:

$$p\left(occ_{x,y}|s^{(1)}, s^{(2)}, \ldots, s^{(T)}\right) =$$

$$1 - \left(1 + \frac{p(occ_{x,y}|s^{(1)})}{1 - p(occ_{x,y}|s^{(1)})} \prod_{\tau=2}^{T} \frac{p(occ_{x,y}|s^{(\tau)})}{1 - p(occ_{x,y}|s^{(\tau)})} \frac{1 - p(occ_{x,y})}{p(occ_{x,y})}\right)^{-1} \quad (1)$$

$p\left(occ_{x,y}|s^{(1)}, s^{(2)}, \ldots, s^{(T)}\right)$ is the combined probability of occupancy of cell (x, y) using sonar measurements from time 1 to T. $p(occ_{x,y})$ is the prior probability which if set to 0.5 the last fraction can be left out.

Pose estimation is omitted because the main focus of the research was creation of the occupancy map. Instead of self-localization a GPS is used for the determination of robot position.

Global grid is built in a merging process of local information. On one hand this process means the transformation of polar coordinates to Cartesian. On the other hand it is the moment of the sensor integration.

Figure 6 shows the occupancy grid of a maze during the process of the exploration.

3.1 Value iteration

After the robot is ready to create a map of its environment, a driving force is needed to urge the robot to explore all the reachable places, otherwise it would wander randomly. For this reason a variant of value iteration is implemented. This technique is well-known in the domain of reinforcement learning [13].

The selected algorithm helps to find the minimum cost-path to unexplored regions of the occupancy grid. A cost matrix is calculated iteratively and after convergence for every occupancy grid cell the cost of travelling to an unexplored grid cell from the actual cell is given.

The first step of the method is the initialization. Naturally the cost of unexplored cells – where occupancy value has not been changed – is 0, while the cost

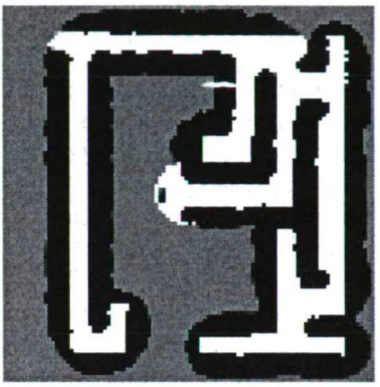


Figure 6: Occupancy grid of a maze

of explored cells is ∞. During the update loop the value of every explored cell is recalculated. Cells with high occupancy probability get 1 as cost. For others the modification is based on a minimum search in the vicinity, the cost together with the probability of occupancy of the neighbouring cells determine the new value of the investigated cell. The ε component is necessary to punish the length of the path.

$\forall$ explored (x, y):

$$V_{x,y} \leftarrow \begin{cases} 1 & \text{if } p\left(occ_{x+i,y+j}\right) > 1 - \delta \\ min(1, \min\limits_{\substack{i=-1,0,1 \\ j=-1,0,1}} \{V_{x+i,y+j} + p\left(occ_{x+i,y+j}\right)\} + \varepsilon) & \text{otherwise} \end{cases}$$

Figure 7 and Figure 8 shows the results of value iteration.

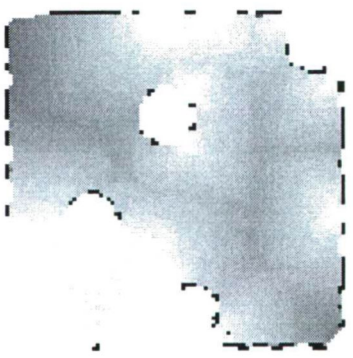


Figure 7: Cost matrix of the open area

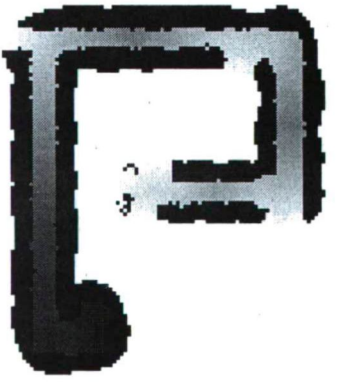


Figure 8: Cost matrix of the maze

Exploration direction is then a resultant of the cost matrix, the actual direction of the robot, and an obstacle-avoidance behaviour.

4 Building a topological graph from occupancy grid

Exploration using value iteration is a very time-consuming task. Values of the cells of the cost matrix are calculated by a process which scans through the whole matrix many times. Furthermore the next exploration direction is based on this gradient map and it does not necessarily take into account the constraint of the robot dynamism, sometimes resulting a fairly clumsy movement.

Accordingly it seems a natural improvement to replace the value iteration module with a topological graph. The topological graph emphasizes the links between landmarks, the possibility to move from one place to another. Graph edges represent traversable corridors of the environment and graph nodes are the crossings or end points. Navigation using the graph is much faster since its size is some order of magnitude smaller than of the cost matrix. Chapter 2 of my book [5] compares metric and topological navigation in detail.

There are quite many different ways of creating a navigation graph using a metric map. Skeletonization, calculating Voronoi-diagrams, matching opposite contours, sparse pixel approaches are among the possibilities [4, 14]. In any case the occupancy grid can be viewed as a two-dimensional greyscale image of the environment, hence digital image processing methods are valid approaches [12].

According to [15] there are enough efficient digital image processing algorithms so for basic tasks we should not reinvent them or design new ones, this is why I use well-known procedures.

Another attitude what I consider important is that the main goal of vectorization is not to produce highest but acceptable quality vectors in the shortest amount of time.

Since I selected skeletonization, steps of the creation of topological navigation using the occupancy grid are the following:

- skeletonization
- chaining the skeleton to form edges
- graph optimization
- navigation with the graph

4.1 Skeletonization

I decided to produce the skeleton of the explored and unoccupied region of the environment. At the end of the process skeleton points are those places where the robot is hopefully not blocked by any obstacles and can reach all regions of the terrain.

For this reason I utilized medial axis transform (MAT) [16]. An interior point of the shape belongs to the medial axis if this point lies at the same distance from two or more nearest contour points. Unfortunately one drawback of MAT appeared during my tests: medial axis of discrete objects and shapes – like the discrete occupancy grid to be projected – may be disconnected. This deficiency is

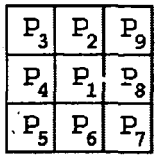


Figure 9: Labeling of points in thinning

not acceptable in our case since the resulting skeleton has to contain all connected routes among important places of the environment.

As a second attempt, instead of using medial axis transform, I applied a thinning algorithm to "peel the union", in other words I iteratively shrank the object to its one pixel wide skeleton [17]. During this process the border pixels are deleted successively while topology and morphology of the object is preserved, that is to say no pixels are deleted at the end of a line or at the connection of two regions.

The thinning algorithm works as it is described in Algorithm 1. Figure 9 shows the labeling of pixels around P_1.

Algorithm 1 The thinning algorithm

$Z0(P_1)$ - the number of zero to nonzero translations in the sequence $\{P_2,P_3,P_4,P_5,P_6,P_7,P_8,P_9,P_2\}$
$NZ(P_1)$ - the number of nonzero neighbours of P_1
Steps:
1. Scan through all the points of the image.
2. Calculate $Z0(P_1)$, $NZ(P_1)$, $Z0(P_2)$, $Z0(P_4)$, for all points.
3. Delete P_1 if the following conditions simultaneously satisfied:

$$2 <= NZ(P_1) <= 6,$$
$$Z0(P_1) = 1,$$
$$P_2 * P_4 * P_8 = 0 \text{ or } Z0(P_2) \neq 1$$
$$P_2 * P_4 * P_6 = 0 \text{ or } Z0(P_4) \neq 1$$

Figure 10 and Figure 11 are examples of the result of the skeletonization process using the thinning algorithm.

4.2 Chaining

Navigation on the skeleton of the explored and unoccupied territory is possible and can be more effective than the calculation of the cost matrix of the value iteration because thinning results a data compression. Nevertheless it is advisable to use the skeleton as a basis for further processing.

Skeleton of the explored region is a set of pixels, this structure can be transformed to a graph. First of all, those points have to be determined where skeleton branches meet. These pixels are the crossing points of corridors. After I have se-

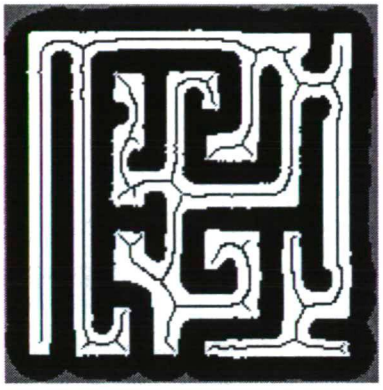


Figure 10: Skeleton of a maze

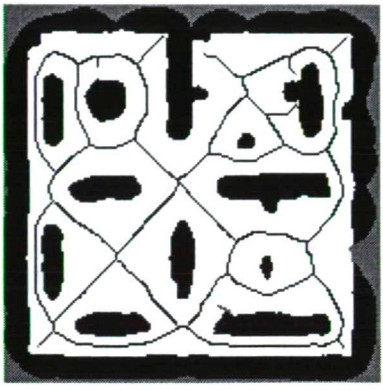


Figure 11: Skeleton of an office

lected the crossing points I cycle through the skeleton branches. This procedure issues in chains, what are pixel sequences from crossing point to crossing point or from crossing point to skeleton end point [14]. Algorithm 2 reveals the main structure of the procedure.

Algorithm 2 Excerpt of the chaining algorithm

```
while there are nodes left do
  c = newChain()
  while there are non-null neighours left do
    if not found getNonNode4Neighbour(q) then
      if not found getNode4Neighbour(q) then
        if not found getNonNode8Neighbour(q) then
          if found getNode8Neighbour(q) then
            append(q,c)
          end
          endChain(c)
        else
          append(q,c)
        end
      else
        append(q,c)
        endChain(c)
      end
    else
      append(q,c)
    end
  end
end
```

The first draft of the graph is calculated during the chaining process. Skeleton crossing point and end points take part in the graph as nodes. Graph edges connect those nodes between which a chain exists.

During my investigation it turned out that the cited algorithm has two minor

problems that, in special cases, corrupts the graph. On Figure 12 and Figure 13 chain creation starts from nodes (marked by 'o') and cycles through all the neighbours of the node (marked by 'x'). Non-node elements are cancelled after they take part in a chain.

First problem rises in situations similar to the one shown on Figure 12. Pixel x marked by 1 (x-1) is cancelled during the chain creation starting from x-2. In the next step – since all the neighbours of nodes have to be processed – chain creation tries to start from an already cancelled node: x-1.

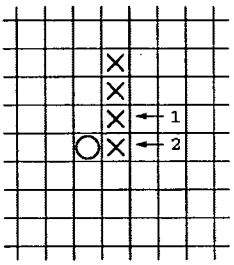


Figure 12: Chaining problem 1

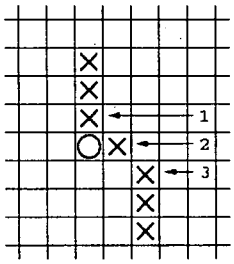


Figure 13: Chaining problem 2

Another problem is indicated on Figure 13. If the chain creation starts from x-2 then in the next step the search should turn to x-3 and chain the pixels downwards. However there is no explicit constraint in the algorithm to prevent the continuation after x-2 in the direction of x-1, what is obviously wrong, since it leaves x-3 without a connection to the node. After I corrected these mistakes the chaining algorithm created the draft of the navigation graph.

4.3 Graph optimization

First version of the graph is not applicable to navigate because chains may ramble far away from edges and if the robot simply follows the way of an edge it could meet with obstacles.

To cope with this problem it is possible to recursively split the edge in question and ensure that the new particles track the slues of the chain better. There are two different algorithm-family for this approximation.

Wall and Danielsson calculate the area of the surface between the edge and the chain [18]. The iterative computation is performed by determining the sum of successive triangles. If the size of the surface exceeds a certain threshold then splitting of the edge is necessary.

Rosin and West's algorithm measures the maximal distance between the edge and the chain [19]. This method splits the edge at its maximum deviation point recursively until all the created new edges are acceptable approximations of the chain (Figure 14).

As a comparison of the methods [14] states that Wall and Danielsson can be implemented very efficiently but on the other hand it is less accurate than Rosin

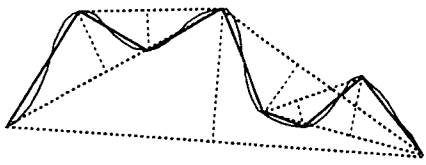


Figure 14: Splitting (taken from the slides of [14])

and West's method. Additionally the second mentioned algorithm may split up edges into small pieces near junctions.

Since I would like to use the topological graph for navigation at the end, it is important that edges do not cross or reach obstacles and walls. In other words fidelity of the graph to the calculated chain is important so I have chosen and implemented Rosin and West's algorithm. The procedure is described in Algorithm 3.

Algorithm 3 Algorithm of Rosin and West

```
split_edge(graph,start_point,end_point) {
  while chain is not finished do
    get_act_point(chain,act_point)
    h = height(start_point,end_point,act_point)
    if h > LIMIT then
       delete_edge_from_graph(graph,start_point,end_point)
       add_node_to_graph(graph,act_point)
       add_edge_to_graph(graph,start_point,act_point)
       add_edge_to_graph(graph,act_point,end_point)
       split_edge(start_point,act_point)
       split_edge(act_point,end_point)
    end
  end
}
```

When the recursive splitting is finished, pruning of edges is useful especially near to unexplored regions. Otherwise, if the robot simply moves to an end node where unexplored territory is nearby, then accidentaly it could run into a wall.

Figure 16 shows the optimized graph of Figure 15 after recursive edge splitting and pruning.

4.4 Navigation

When creation of the graph of the explored and not occupied region is complete, the robot has to determine the next exploration direction. Generally the robot is aimed to sweep through all the reachable places of the environment. This is why those nodes of the graph where unexplored region is close can be considered as goal nodes.

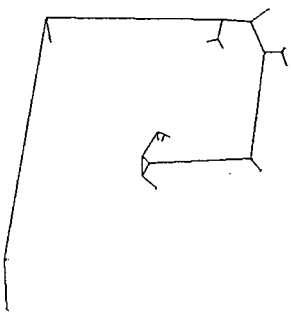


Figure 15: Graph after chaining

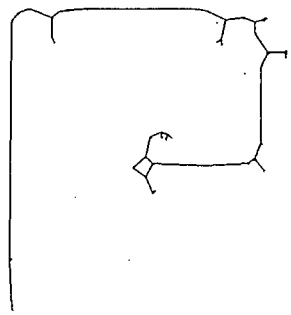


Figure 16: Optimized graph

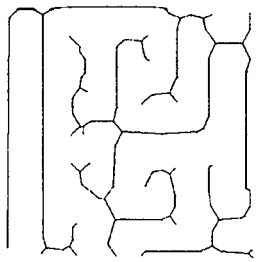


Figure 17: Navigation graph of the maze

To localize these elements I performed a general A^* algorithm [20]. This classical algorithm finds the shortest path from the predefined start node of the graph to a goal node. Start node of the graph in our case is the actual position of the robot. The A^* algorithm then calculates the shortest path from the actual position to a node where exploration could be fruitful.

Using the shortest path as a list to be processed, the robot can turn to the next node of the graph in the list and move directly ahead while it does not reach the last node in the list.

Besides the topological graph and the A^* algorithm the final robot movement is comprised another behaviour pattern as well. The role of this normal move module is to stimulate the robot straight ahead on clear sights, and it also ensures obstacle avoidance motion in case of necessity. Since the generation of the topological graph is time-consuming, this job is not done continuously. When the normal move module does not explore efficiently, in other words the explored surface does not grow enough, creation of the graph takes place and navigation is governed by the A^* algorithm. This alternating comportment incorporates the advantages of the two behaviour modules.

5 Results

The navigation algorithms were tested in five different environments in several experiments from various starting points. The environments were selected to cover a wide range of possible situation that could arise during map-building.

The terrains were the following: an open area with some round obstacles, a radial maze taken from [21] well-known in cognitive map researches (Figure 4), a maze (Figure 2), an office-like room (Figure 3) which was one of the fields of the Artificial Life Creators Contest, and a labyrinth used at the 1994 AAAI autonomous mobile robot competition (Figure 5, [4]).

The open area is 1 m^2, the AAAI maze is 1.85 m^2, while the others are 2.25 m^2. Five attempts were performed in every field with both algorithm. The robot could explore all the environments by the two methods.

In the small and easily solvable open area the robot spends 8 and 6.4 minutes on an average in robot performance time using value iteration and topological graph respectively. Radial maze does not cause any difficulties for the two programs, both solves it in around 6 minutes on an average.

The most significant advance can be reached in the office environment: the 20 minutes time drops to 12.4 minutes. In the maze the time profit is smaller: the 22 minutes of value iteration is reduced to 14.5 minutes. The AAAI contest environment is easier to solve than the maze, hence time frames of value iteration and graph navigation are 14 and 11.7 respectively.

These results are collected in Table 1.

Table 1: Time comparison of the navigation methods

	Value iteration (min)	**Topological graph (min)**
Open room	8	6.4
Radial	6.3	6
Office	20	12.4
Maze	22	14.5
AAAI contest	14	11.7

The acceleration between the two methods is a consequence of the smaller number of entities with which the algorithms have to deal (Table 2). There are between 11600 and 28900 pixels in the cost matrix of the value iteration, and the number of graph nodes are between 20 and 120, depending on the size and the complexity of the environment.

6 Conclusions

This paper presents a method to build a topological graph for navigation based on occupancy grid in the simulation environment of Webots. Besides the fact that

Table 2: Number of entities in the navigation methods

	Value iteration (pixels)	**Topological graph (graph nodes)**
Open room	12800	50
Radial	11600	20
Office	28900	110
Maze	28900	120
AAAI contest	23700	105

already known algorithms are used, significantly better accomplishments related to the pure occupancy grid method justify this navigation approach.

Using topological graph instead of value iteration for the determination of exploration direction seems a beneficial modification. On one hand it approximates better the nature of the navigation. On the other hand the new algorithm performs better.

First of all, the number of manipulated entities – pixels for the value iteration, and graph nodes for the topological navigation – differ in the two approaches. This gap is more than two orders of magnitude, so the graph navigation dramatically reduces the need for resources, especially the need for memory.

Secondly, better total exploration time can be achieved with the newer control procedure. Differences in the acceleration among various test fields follow from the fact that the graph mostly helps in elongated parts of the territory and at the connections of the large spaces. Open spaces are easily explorable by random obstacle avoidance so the necessary time for open room and radial maze is not diminished essentially. For the maze, the office, and the AAAI contest environment the effects are easily recognizable, since time profit exceeds 20%.

7 Future work

There are quite many different ways of continuing the research. Some of them are mentioned below:

- Testing the algorithms in real robot.
- Higher level task can be performed by the robot after successful exploration.
- Moving around in dynamic environments is a serious challenge, this extension would make the problem more interesting.
- Using pose estimation may make the robot fully automate.
- Introduction of new sensor types especially video cameras may enhance the occupancy grid creation and position estimation as well.

Acknowledgement
The author wishes to thank György Kampis for his useful suggestions, and András Salamon for his valuable remarks.

References

[1] O. Michel. Professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, 1(1):39–42, 2004.

[2] S. Thrun. Probabilistic algorithms in robotics. *AI Magazine*, 21(4):93–109, 2000.

[3] R. Szabó. Navigation of simulated mobile robots in the webots environment. *Periodica Polytechnica — Electrical Engineering*, 47(I-II):149–163, 2003.

[4] S. Thrun. Learning metric-topological maps for indoor mobile robot navigation. *Artificial Intelligence*, 99(1):21–71, 1998.

[5] R. Szabó. *Mobil robotok szimulációja.* Eötvös Kiadó, 2001.

[6] S. Thrun. Robotic mapping: A survey. In G. Lakemeyer and B. Nebel, editors, *Exploring Artificial Intelligence in the New Millenium.* Morgan Kaufmann, 2002. to appear.

[7] Darpa grand challenge, 2004. http://www.darpa.mil/grandchallenge04/.

[8] T. M. Mitchell. *Machine learning.* McGraw Hill, 1997.

[9] G. Welch and G. Bishop. An introduction to the kalman filter. Technical report, University of North Carolina, Department of Computer Science, Chapel Hill, NC, USA, 2003.

[10] S. Thrun, W. Burgard, and D. Fox. A probabilistic approach to concurrent mapping and localization for mobile robots. *Machine Learning and Autonomous Robots*, 31(5):1–25, 1998.

[11] H. P. Moravec and A. Elfes. High resolution maps from wide angle sonar. In *Proceedings of the IEEE International Conference on Robotics and Automation, (St. Louis, MO)*, pages 116–121, 1985.

[12] A. Elfes. Using occupancy grids for mobile robot perception and navigation. *Computer*, 22(6):46–57, 1989.

[13] R. Sutton and A. Barto. *Reinforcement Learning: An Introduction.* Bradford Book, MIT Press, 1998.

[14] Karl Tombre, Christian Ah-Soon, Philippe Dosch, Gérald Masini, and Salvatore Tabbone. Stable and robust vectorization: How to make the right choices. *Lecture Notes in Computer Science*, 1941:3–17, 2000.

[15] K. Tombre, C. Ah-Soon, P. Dosch, A. Habed, and G. Masini. Stable, robust and off-the-shelf methods for graphics recognition, 1998.

[16] G. Borgefors. Distance transformations in digital images. *Computer Vision, Graphics, and Image Processing*, 34:344–371, 1986.

[17] A. K. Jain, editor. *Fundamentals of Image Processing.* Prentice-Hall,NJ, 1989.

[18] K. Wall and P.-E. Danielsson. A fast sequential method for polygonal approximation of digitized curves. *Computer Vision, Graphics and Image Processing,* 28:220–227, 1984.

[19] P. L. Rosin and G. A. West. Segmentation of edges into lines and arcs. *Image and Vision Computing,* 7(2):109–114, 1989.

[20] Futó Iván, editor. *Mesterséges intelligencia.* Aula Kiadó, 1999.

[21] V. Csányi. *Etológia.* Nemzeti Tankönyvkiadó Rt., 1994.

Acta Cybernetica **17** (2005) 419–445.

Extending the Sparkle Core language with object abstraction*

Máté Tejfel*, Zoltán Horváth*, and Tamás Kozsik†

Abstract

Sparkle is a theorem prover specially constructed for the functional programming language Clean. In a pure functional language like Clean the variables represent constant values; variables do not change in time. Hence it seems that temporality has no meaning in functional programs. However, in certain cases (e.g. in interactive or distributed programs, or in ones that use I/O), a series of values computed from one another can be considered as different states of the same "abstract object". For this abstract object temporal properties can be proved. This paper presents a method to describe abstract objects and invariant properties in an extended version of the Sparkle Core language. The creation of such descriptions will be supported by a refactoring tool. The descriptions are completely machine processible, and provide a way to automatize the proof of temporal properties of Clean programs with the extended Sparkle system.

Categories and Subject Descriptors: D.1.1 [Programming Techniques]: Applicative (Functional) Programming; F.3.1 [Logics and meanings of programs]: Specifying and Verifying and Reasoning about Programs - *invariants*;

Key Words and Phrases: Verification, invariant properties, abstract functional object, Clean, Sparkle

1 Introduction

The temporal logical operators describe how the values of the program variables (the so-called program state) vary in time. They are very useful for proving correctness of (sequential or parallel) imperative programs. Some well-known such operators are e.g. "nexttime", "sometimes", "always" and "invariant". All these operators can be expressed based on the "weakest precondition" operator [7, 13].

The weakest precondition of a program statement with respect to a postcondition holds for a state "a" if and only if the statement starting from "a" always

*Supported by the Hungarian National Science Research Grant (OTKA), Grant Nr.T037742. and by the Bolyai Research Scholarship.

†Department of Programming Languages and Compilers Eötvös Loránd University, Budapest, e-mail: matej@inf.elte.hu, hz@inf.elte.hu, kto@inf.elte.hu

terminates in a state for which the postcondition holds. We can compute the weakest precondition of a statement in an automated way: we have to rewrite the postcondition according to the substitution rules defined by the statement.

When proving correctness of functional programs, the practicability of temporal operators is not obvious. In a pure functional programming language a variable is a value, like in mathematics, and not an "object" that can change its value in time, viz. during program execution. Due to referential transparency, reasoning about functional programs can be accomplished with a fairly simple mathematical machinery, using, for example, classical logic and induction (see [22]). This fact is one of the basic advantages of functional programs over imperative ones.

However, in certain cases it is natural to express our knowledge about the behaviour of a functional program (or rather our knowledge about the values the program computes) in terms of temporal logical operators. Moreover, in the case of parallel or distributed functional programs, temporal properties are exactly as useful as they are in the case of imperative programs. For example, those invariants which are preserved by all components of a distributed or parallel program, are also preserved by the compound program.

In the authors' approach, certain values computed during the evaluation of a functional program can be regarded as successive values of the same "abstract object". This corresponds directly to the view which certain object-oriented functional languages hold.

Clean [24], a lazy, pure functional language was chosen for this research. An important factor in our choice was that a theorem prover, Sparkle [22] is already built in the integrated development environment of Clean. Sparkle supports reasoning about Clean programs almost directly. The authors extended the basic logic used by Sparkle with temporal operators.

Earlier, correctness proofs about interactive, concurrent (interleaved) Clean programs, namely Object IO processes have been provided in [14, 15]. However, these proofs were carried out by hand. The authors argue that the extension of the theorem prover with tools supporting temporal logical operators facilitates the reasoning about interactive, concurrent or distributed (see [16]) Clean programs, since temporal logical reasoning can be performed within the theorem prover.

For formulating and proving temporal properties of a Clean program, the "abstract objects" have to be determined, that is it has to be specified which functional (mathematical) values correspond to different states of the same abstract object. Furthermore, state transitions should also be expressible. Therefore, Clean and correspondingly the Sparkle Core language have to be extended with some new syntactical elements. This paper aims to present these extensions, to show how one can describe abstract objects in these extended languages, to give the semantics of the introduced language extensions, and to illustrate by means of some simple examples that temporal reasoning is really useful for functional programs.

The rest of the paper is organized in the following way. In Section 2 the object abstraction method is presented through a simplistic example. Then Section 3 describes an extension to the Clean language. Section 4 introduces the new language constructs into the Sparkle Core language. Section 5 explains how temporal

propositions can be expressed in the extended Sparkle framework. Next, Section 6 presents some more complex and more useful examples of temporal properties and their proofs. Finally, in Section 7, the conclusions are drawn and future work is defined.

2 Object abstraction

In Clean the uniqueness type system makes destructive updates possible without violating referential transparency. In the case of unique values temporality has a similar meaning as in imperative languages: unique values encode states. Destructive updates are not merely used to increase the efficiency of Clean programs, but the I/O system of Clean is also defined in terms of state transitions over a "unique environment". (The other well-known technique to define I/O in a pure functional language is the monadic approach, applied in the language Haskell [23].)

Programs written with the Object I/O library (a standard API for Clean) [1] are reactive. They create a unique state space and define initialisation and state transition functions. The library supports *interactive processes*, which can be created and closed dynamically. Each interactive process may consist of an arbitrary number of *interactive objects*. Since I/O processes may run in parallel (in an interleaved manner), their behaviour can be described with a temporal logical machinery [4, 13] (in contrast to e.g. [3, 10, 11]). The authors have researched this issue in [14, 15].

This paper exploits a more general method, discussed before in [17, 18]. In this methodology one can reason about temporal properties of Clean programs even if they do not use unique values or interactive Object I/O processes. Not only some call-back functions of Object I/O can be state transition functions: the programmer can demarcate state transitions explicitly in a more flexible way. Different values computed by a functional program and stored in variables (in the functional sense of variables) can be regarded as different states of the same object. A state transition will thus be a piece of functional code that computes such a value from another one.

The following simple example introduces shortly the object abstraction method. The example program `sort3` puts three integer values in increasing order. It uses another function `sort2`, which, in turn, puts two integer values in increasing order.

```
sort3 a b c
   # (a, b) = sort2 a b
   # (b, c) = sort2 b c
   # (a, c) = sort2 a c
   = (a, b, c)
```

According to the scoping rules in Clean, this program is equivalent to the following one:

```
sort3 a1 b1 c1
   # (a2, b2) = sort2 a1 b1
   # (b3, c2) = sort2 b2 c1
   # (a3, c3) = sort2 a2 c2
   = (a3, b3, c3)
```

Here the values a_1, a_2 and a_3 may be associated to the same abstract object, e.g. obj_1. Similarly, the values b_i and c_i may be associated to obj_2 and obj_3, respectively. The let-before expressions (denoted by #) will hence become the state transitions (atomic actions) of this program. Clean has to be extended with new syntactical elements so that one can express this kind of "object abstraction".

Although this example may seem too simplistic, it illustrates well the technique used for introducing objects. For more complex examples the same technique can be used, but the propositions may become less readable and the proofs substantially longer. To address the first issue, the authors are planning to develop a tool that provides support for object abstraction on a graphical user interface. Managing the second issue requires the use of predefined libraries of domain-specific lemmas.

3 Extending Clean with object abstraction

For the demarcation of "abstract objects" two new language constructs are needed. One of the constructs will be used to define which values (functional variables) correspond to different states of the same abstract object. The other construct will mark the state transitions of the program: in each state transition, one or more objects may change their values. State transitions will be regarded as atomic actions with respect to the temporal logical operators, and will be referred to as "steps" in the forthcoming sections.

The first construct will be denoted by ".|.". It has two arguments: an object identifier and a value identifier, like in " .|. `object_id value_id` ". This means that the value identified by `value_id` is associated to the abstract object identified by `object_id`, or, for short, `value_id` identifies a state of `object_id`. The second construct, used for marking steps, is similar to the let-before (#) construct of Clean, hence a similar syntax has been chosen for that: ".#.".

The Clean syntax has been extended with the two constructs in the following way. The original definition of `Variable` has been changed to include an alternative "`Object`".

```
Variable = LowerCaseId
         | Object

Object = .|. LowerCaseId LowerCaseId
```

Therefore, an `Object` can be used wherever a `Variable` can be used in (the original) Clean, under the following conditions. A function definition may introduce objects, only if the body of the function is made up of let-before constructs. (The current

implementation of the extended Sparkle system can handle multiple function alternatives, but does not allow objects in functions with guards and rule alternatives.) The objects are local to the function definition, and the same object name refers to the same object in this scope. (Defining multiple objects with the same name within the same function definition is disallowed.) Objects can only be used in "steps". In every binding within a step (a so called *StepBind*, see later), the same object can appear at most once on the left, and at most once on the right-hand side. Obviously, the variables constituting the states of an abstract object must be of the same type. Finally, it has to be noted that currently only objects within a single function definition are supported: the variables that make up the states of the object must be defined in the same function. (In the near future the authors plan to develop an enhanced version of extended Clean and extended Sparkle in which object abstraction is not restricted to happen within the boundaries of a single function.)

The rule for `LetBeforeExpression` has also been extended with a new alternative, `StepExpression`, to support the second introduced new construct. A `StepExpression` can be used in the extended Clean wherever a `LetBeforeExpression` can be used in Clean.

```
LetBeforeExpression = # {GraphDef}+
                    | #!{GraphDef}+
                    | StepExpression

StepExpression = .#. {GraphDef}+
```

The example program expressed in the extended Clean language is the following.

```
sort3 (.|. obj1 a1) (.|. obj2 b1) (.|. obj3 c1)
   .#. ((.|. obj1 a2), (.|. obj2 b2)) = sort2 (.|. obj1 a1) (.|. obj2 b1)
   .#. ((.|. obj2 b3), (.|. obj3 c2)) = sort2 (.|. obj2 b2) (.|. obj3 c1)
   .#. ((.|. obj1 a3), (.|. obj3 c3)) = sort2 (.|. obj1 a2) (.|. obj3 c2)
    =  ((.|. obj1 a3), (.|. obj2 b3), (.|. obj3 c3))
```

This program text is much harder to read than the original Clean program. Note, however, that the extended Clean language is just an intermediate language used between two programs. A refactoring tool [25], integrated into an interactive software development environment, will be used to produce extended Clean code. The input to the refactoring tool is the original Clean code, and *interactive* instructions from the programmer regarding which values belong to which objects. The extended Clean code will be processed by a proof system, namely an extended version of Sparkle.

The semantics of Clean can be expressed in terms of Sparkle Core [21]. The Sparkle Core language is a part of the formal framework used by the Sparkle theorem prover, which has been developed for reasoning about Clean programs. Essentially, Sparkle Core corresponds to the internal representation of Clean programs in the Clean compiler. It is possible to express the semantics of the extended Clean

language in terms of a *variant* of Sparkle Core. The next sections present how Sparkle Core was modified to support extended Clean, and how Sparkle is to be modified to enable formulating and proving invariants of abstract objects.

4 Extending the Sparkle Core language

In order to adapt Sparkle for reasoning about temporal properties of abstract objects, the constructs `.|.` and `.#.` of the extended Clean language have been introduced into Sparkle Core. Here an incomplete description of this extended version of Sparkle Core is provided for the interested Reader, focusing only on the new elements added to Sparkle Core. An informal explanation of these new elements are also given, but for further details on the formal syntax and semantics of Sparkle Core the Reader is referred to [21]. (For the sake of readability, at certain points the notations of Sparkle Core have been simplified.)

The expressions ($\mathcal{E}$) of Sparkle Core are made up of variables, basic values, (function, delta-rule and constructor) symbols, applications, case-expressions, (lazy and strict) let-expressions, and (typed) "undefined" expressions. Two more alternatives have been introduced: object definitions and steps. Object definitions associate object identifiers (from $\mathcal{O}$) to expression-variable identifiers (from $\mathcal{V}_e$). Steps are similar to let expressions, they contain bindings of local expression variables ($\mathcal{V}_e^\bullet$) and objects to expressions. The resulting definition of expressions, $\mathcal{E}^{ext}$, is as follows. (The new elements are framed.)

$$
\begin{aligned}
\mathcal{E}^{ext} = & \{\, \texttt{var}\ x \mid x \in \mathcal{V}_e \} \\
& \cup \boxed{\mathcal{O}^{\texttt{def}}} \\
& \cup \{\texttt{basic}\ b \mid b \in \mathcal{B}_v\} \\
& \cup \{\texttt{symbol}\ s\ \sigma s\ es \mid s \in \mathcal{S}^e,\ \sigma s \in \langle \mathcal{T} \rangle,\ es \in \langle \mathcal{E}^{ext} \rangle \\
& \qquad\qquad \mid |\sigma s| = Arity^1(s) \wedge |es| \leq Arity^2(s)\} \\
& \cup \{\texttt{apply}\ e_1\ \texttt{to}\ e_2 \mid e_1 \in \mathcal{E}^{ext}, e_2 \in \mathcal{E}^{ext}\} \\
& \cup \{\texttt{case}\ e\ \texttt{of}\ alts \mid e \in \mathcal{E}^{ext}, alts \in \langle Alt \rangle\} \\
& \cup \{\texttt{let}\ binds\ \texttt{in}\ e \mid binds \in \langle LetBind \rangle,\ e \in \mathcal{E}^{ext}\} \\
& \cup \boxed{\{\texttt{step}\ stepbinds\ \texttt{in}\ e \mid stepbinds \in \langle StepBind \rangle,\ e \in \mathcal{E}^{ext}\}} \\
& \cup \{\texttt{let!}\ x_1 = e_1\ \texttt{in}\ e_2 \mid x \in \mathcal{V}_e^\bullet,\ e_1 \in \mathcal{E}^{ext},\ e_2 \in \mathcal{E}^{ext}\} \\
& \cup \{\bot_\sigma \mid \sigma \in \mathcal{T}\}
\end{aligned}
$$

$$
\begin{aligned}
StepBind &= \{x\ \texttt{binds}\ e \mid x \in \mathcal{V}_e^\bullet \cup \mathcal{O}^{def},\ e \in \mathcal{E}^{ext}\} \\
\mathcal{O}^{\texttt{def}} &= \{\texttt{obj}\ o\ x \mid o \in \mathcal{O},\ x \in \mathcal{V}_e\} \\
\mathcal{O} &= \{\texttt{objid}\ z \mid z \in \mathbf{Z}\}
\end{aligned}
$$

Note that in *StepBind* not only object states can be bound, but (local) variables as well ($x \in \mathcal{V}_e^\bullet$). There is a technical reason for that: this is how the "current state" of an object can be retrieved and used in an expression.

The changes to $\mathcal{E}$ induce further modifications, e.g. in the definition of functions ($\mathcal{F}^{\mathtt{def}}$) in Sparkle Core. The modifications make it possible to use objects both in the formal parameter list and in the body of functions, granted that the same object identifier does not occur more than once in the formal parameter list.

As an illustration, the two definitions of the function `sort3` (expressed in Clean, in Section 2 and in extended Clean, in Section 3) turned into Sparkle Core and extended Core are provided. It is instructive to see the differences between the two definitions, without trying to understand all their details. First, let us have a look at the one without objects. (Almost the same information is available in Sparkle – using the appropriate options offered by its user interface – as in the Sparkle Core code below.)

```
fundef sort3 αs σs1 τ ⟨(var (exprvar 1)),
                       (var (exprvar 2)),
                           (var (exprvar 3)))⟩
  let ⟨ (exprvar 4)
          binds (symbol sort2 σs2 ⟨(var (exprvar 1)) (var (exprvar 2)))⟩ ),
      (exprvar 5) binds (symbol _tupleselect_2_1 σs3 ⟨ (var (exprvar 4)) ⟩ ),
      (exprvar 6) binds (symbol _tupleselect_2_2 σs3 ⟨ (var (exprvar 4)) ⟩ ),
      (exprvar 7)
          binds (symbol sort2 σs2 ⟨ (var (exprvar 6)), (var (exprvar 3)) ⟩ ),
      (exprvar 8)
          binds (symbol _tupleselect_2_1 σs3 ⟨ (var (exprvar 7)) ⟩ ),
      (exprvar 9)
          binds (symbol _tupleselect_2_2 σs3 ⟨ (var (exprvar 7)) ⟩ ),
      (exprvar 10)
          binds (symbol sort2 σs2 ⟨ (var (exprvar 5)), (var (exprvar 9)) ⟩ ),
      (exprvar 11)
          binds (symbol _tupleselect_2_1 σs3 ⟨ (var (exprvar 10)) ⟩ ),
      (exprvar 12)
          binds (symbol _tupleselect_2_2 σs3 ⟨ (var (exprvar 10)) ⟩ ) ⟩
  in symbol tuple3 σs4 ⟨(var(exprvar 11)),
                        (var(exprvar 8)),
                            (var (exprvar 12)))⟩
```

Compare the above definition with the following *extended* Sparkle Core code:

```
fundef sort3 αs σs1 τ ⟨(obj (objid 1) (exprvar 1)),
                       (obj (objid 2) (exprvar 2)),
                           (obj (objid 3) (exprvar 3)))⟩
  step ⟨ (exprvar 4)
           binds (symbol sort2 σs2 ⟨ (obj (objid 1) (exprvar 1)),
                                     (obj (objid 2) (exprvar 2)) ⟩ ),
         ( obj (objid 1) (exprvar 5))
           binds (symbol _tupleselect_2_1 σs3 ⟨ (var (exprvar 4)) ⟩ ),
         ( obj (objid 2) (exprvar 6))
```

```
          binds (symbol _tupleselect_2_2 σs3 ⟨ (var (exprvar 4)) ⟩ ) )
in step ⟨ (exprvar 7)
            binds (symbol sort2 σs2 ⟨ (obj (objid 2) (exprvar 6)),
                                      (obj (objid 3) (exprvar 3)) ⟩ ),
          ( obj (objid 2) (exprvar 8))
            binds (symbol _tupleselect_2_1 σs3 ⟨ (var (exprvar 7)) ⟩ ),
          ( obj (objid 3) (exprvar 9))
            binds (symbol _tupleselect_2_2 σs3 ⟨ (var (exprvar 7)) ⟩ ) )
in step ⟨ (exprvar 10)
            binds (symbol sort2 σs2 ⟨ (obj (objid 1) (exprvar 5)),
                                      (obj (objid 3) (exprvar 9)) ⟩ ),
          ( obj (objid 1) (exprvar 11))
            binds (symbol _tupleselect_2_1 σs3 ⟨ (var (exprvar 10)) ⟩ ),
          ( obj (objid 3) (exprvar 12))
            binds (symbol _tupleselect_2_2 σs3 ⟨ (var (exprvar 10))⟩ ) )
in symbol tuple3 σs3 ⟨(obj (objid 1) (exprvar 11))
                      (obj (objid 2) (exprvar 8))
                        (obj (objid 3) (exprvar 12))⟩
```

The semantics of Sparkle Core has been changed in such a way that object definitions and steps are only used during the formulation and proof of temporal properties. Otherwise objects can be reduced to variables and steps to let-expressions. The reduction rules expressing this are the following.

$$\frac{\texttt{obj}\ o\ x}{x} \qquad \frac{\texttt{step}\ \mathit{letbinds}\ \texttt{in}\ e}{\texttt{let}\ \mathit{letbinds}\ \texttt{in}\ e}$$

Note that the "stepbinds" part of a step have to be reduced to a "letbind" (by reducing all objects of a stepbind to variables) before reducing it to a let-definition.

5 Temporal propositions

For formulating temporal properties of abstract objects, the logical framework of Sparkle has to be made capable to manage temporal propositions. This paper shows how safety properties, namely *invariants* and *unless* properties should be handled in this extended framework. The definition $\mathcal{P}$ of propositions has been changed to include temporal propositions. To describe e.g. invariants, $\mathcal{P}^{inv}$ is introduced:

$$\mathcal{P}^{ext} = \mathcal{P} \cup \mathcal{P}^{inv} \cup \ldots$$
$$\mathcal{P}^{inv} = \{p\ \texttt{inv}\ (f\ cxs)\ q \mid q \in \mathcal{P},\ p \in \mathcal{OP},\ f \in \mathcal{F}, cxs \in \langle\mathcal{E}\rangle\}$$

An invariant proposition "p inv $(f\ cxs)\ q$" means that proposition p holds invariantly during the evaluation of "$f\ cxs$" with respect to the precondition q. In the definition above, f is a function symbol, and cxs is an actual parameter list containing expressions of (the original) Sparkle Core. Furthermore, q is a proposition

of the basic logic of (the original) Sparkle, referring to the variables occurring in cxs. On the other hand, p is an "object proposition" ($\mathcal{OP}$), which can refer to object identifiers as well. $\mathcal{OP}$ differs from $\mathcal{P}$ in that the expressions occurring in it come from a modified set of expressions $\mathcal{E}^{temp}$ instead of $\mathcal{E}$. The definition of $\mathcal{E}^{temp}$ adds the alternative $\mathcal{O}$ to $\mathcal{E}$. (Notice the difference between $\mathcal{E}^{temp}$ and $\mathcal{E}^{ext}$. The latter introduces an alternative for $\mathcal{O}^{\mathbf{def}}$, not $\mathcal{O}$, and a further alternative for steps.)

As an example, consider the following invariant property of the `sort3` function. It states that, given the precondition $x + y + z = 0$, the sum of the three objects is equal to 0 during the evaluation of "`sort3 x y z`".

$$\forall x \forall y \forall z \ (obj_1 + obj_2 + obj_3 = 0) \ \mathtt{inv} \ (sort3 \ x \ y \ z) \ (x + y + z = 0)$$

The names identifying the objects (obj_1, obj_2 and obj_3) are declared in the `sort3` function: the object names appearing in an invariant proposition are resolved in the scope of the function the proposition is referring to.

In extended Sparkle Core the above invariant is formulated in the following way:

```
(forall exprs (exprvar 53) (forall exprs (exprvar 54) (forall exprs (exprvar 55)
  ((symbol (+) σs ⟨ (objid 1),
                    (symbol (+) σs ⟨ (objid 2), (objid 3) ⟩) ⟩)
                      equals (basic (int 0)) )

  inv (sort3 ⟨ (var (expvar 53)), (var (exprvar 54)), (var (exprvar 55))))

  ((symbol (+) σs ⟨ (var (exprvar 53)),
          (symbol (+) σs ⟨ (var (exprvar 54)), (var (exprvar 55)) ⟩) ⟩)
                  equals (basic (int 0))) )))
```

Invariants should follow from the precondition, and must be preserved by the (atomic) state transitions [4, 13]. The preservation of a statement with respect to a state transition is expressed with the weakest precondition operator wp [7]. In this case there are three state transitions, corresponding to the three evaluations of `sort2`. Let us abbreviate "$obj_1 + obj_2 + obj_3 = 0$" with p and "$x + y + z = 0$" with q. After introducing (fixing) the universally quantified variables x, y and z, the invariant can be rewritten to the conjunction of the following four proposition.

1. $q \Rightarrow p$
2. $q \wedge p \ \Rightarrow \ wp(obj_1, \ obj_2 \ = \ sort2 \ obj_1 \ obj_2)(p)$
3. $q \wedge p \ \Rightarrow \ wp(obj_2, \ obj_3 \ = \ sort2 \ obj_2 \ obj_3)(p)$
4. $q \wedge p \ \Rightarrow \ wp(obj_1, \ obj_3 \ = \ sort2 \ obj_1 \ obj_3)(p)$

Since the precondition may only refer to variables, and not to objects ($q \in \mathcal{P}$), it can be used as a hypothesis in each generated propositions.

The meaning of the invariant proposition, of course, is given with respect to a program context, e.g. the definition of the function `sort3`. In the appendix the function computing the semantical value of invariant propositions is provided. Applying this semantical function, the semantics of the above invariant is obtained in the Sparkle system.

```
(forall exprs (exprvar 53) (forall exprs (exprvar 54) (forall exprs (exprvar 55)
      binary (binary (binary prop_init and prop_1) and prop_2) and prop_3
)))
```

with the following abbreviations:

```
prop_init =
 (forall exprs (exprvar 56) (forall exprs (exprvar 57) (forall exprs (exprvar 58)
  (binary ((symbol  (+) σs ⟨ (var (exprvar 53)),
             (symbol  (+) σs ⟨ (var (exprvar 54)), (var (exprvar 55)) ⟩) ⟩)
            equals (basic (int 0)))
    implies
    (binary ( binary ( binary ( (var (exprvar 56) ) equals (var (exprvar 53) ) )
      and ( (var (exprvar 57) ) equals (var (exprvar 54) ) ) )
      and ( (var (exprvar 58) ) equals (var (exprvar 55) ) ) )
    implies
      ((symbol  (+) σs ⟨ (var (exprvar 56)),
          (symbol  (+) σs ⟨ (var (exprvar 57)), (var (exprvar 58)) ⟩) ⟩)
      equals (basic (int 0)) ) ) ) )))

prop_1 =
 (forall exprs (exprvar 59) (forall exprs (exprvar 60)
 (forall exprs (exprvar 61) (forall exprs (exprvar 62) (forall exprs (exprvar 63)
  (binary ((symbol  (+) σs ⟨ (var (exprvar 53)),
             (symbol  (+) σs ⟨ (var (exprvar 54)), (var (exprvar 55)) ⟩) ⟩)
            equals (basic (int 0)))
    implies
  (binary ((symbol (+) σs ⟨ (var (exprvar 59)),
                  (symbol(+) σs ⟨ (var (exprvar 60)), (var (exprvar 61)) ⟩) ⟩)
                        equals (basic (int 0)))
  implies
    (binary (binary (binary ( (var (exprvar 4))
      equals ( symbol sort2  σs_2 ⟨ (var (exprvar 59)), (var (exprvar 60) ⟩) ) )
    and ((var (exprvar 62)) equals (symbol _tupleselect_2_1  σs_3
                                        ⟨ (var (exprvar 4))⟩ ⟩) )
    and ((var (exprvar 63)) equals (symbol _tupleselect_2_2  σs_3
                                        ⟨ (var (exprvar 4))⟩ ) ) )
  implies
    ((symbol (+) σs ⟨ (var (exprvar 62)),
            (symbol (+) σs ⟨ (var (exprvar 63)), (var (exprvar 61)) ⟩) ⟩)
                  equals (basic (int 0))) )))
```

```
prop2 =
  (forall exprs (exprvar 64) (forall exprs (exprvar 65)
  (forall exprs (exprvar 66) (forall exprs (exprvar 67) (forall exprs (exprvar 68)
   (binary ((symbol  (+) σs ⟨ (var (exprvar 53)),
              (symbol  (+) σs ⟨ (var (exprvar 54)), (var (exprvar 55)) ⟩) ⟩)
                equals (basic (int 0)))
     implies
   (binary ((symbol (+) σs ⟨ (var (exprvar 64)),
                    (symbol (+) σs ⟨ (var (exprvar 65)), (var (exprvar 66)) ⟩) ⟩)
                        equals (basic (int 0)))
   implies
      (binary (binary (binary ( (var (exprvar 7))
        equals ( symbol sort2  σs2 ⟨ (var (exprvar 65)), (var (exprvar 66) ⟩) ) )
      and ((var (exprvar 67)) equals (symbol _tupleselect_2_1  σs3
                                        ⟨ (var (exprvar 7))⟩ ⟩) )
      and ((var (exprvar 68)) equals (symbol _tupleselect_2_2  σs3
                                        ⟨ (var (exprvar 7))⟩ ) ) )
   implies
      ((symbol (+) σs ⟨ (var (exprvar 64)),
               (symbol (+) σs ⟨ (var (exprvar 67)), (var (exprvar 68)) ⟩) ⟩)
                  equals (basic (int 0))) )))


prop3 =
  (forall exprs (exprvar 69) (forall exprs (exprvar 70)
  (forall exprs (exprvar 71) (forall exprs (exprvar 72) (forall exprs (exprvar 73)
   (binary ((symbol  (+) σs ⟨ (var (exprvar 53)),
              (symbol  (+) σs ⟨ (var (exprvar 54)), (var (exprvar 55)) ⟩) ⟩)
                equals (basic (int 0)))
     implies
   (binary ((symbol (+) σs ⟨ (var (exprvar 69)),
                    (symbol (+) σs ⟨ (var (exprvar 70)), (var (exprvar 71)) ⟩) ⟩)
                        equals (basic (int 0)))
   implies
      (binary(binary (binary ( (var (exprvar 10))
        equals ( symbol sort2  σs2 ⟨ (var (exprvar 69)), (var (exprvar 71) ⟩) ) )
      and ((var (exprvar 72)) equals (symbol _tupleselect_2_1  σs3
                                        ⟨ (var (exprvar 10))⟩ ⟩) )
      and ((var (exprvar 73)) equals (symbol _tupleselect_2_2  σs3
                                        ⟨ (var (exprvar 10))⟩ ) ) )
   implies
      ((symbol (+) σs ⟨ (var (exprvar 72)),
               (symbol (+) σs ⟨ (var (exprvar 70)), (var (exprvar 73)) ⟩) ⟩)
                  equals (basic (int 0))) )))
```

6 Some more complex examples

In this section two more complex examples are presented. The first example introduces a simple program modelling a database of financial transactions, and an invariant property of this program is provided. The second example is an implementation of the dining philosophers' problem, and it is used to illustrate unless properties.

6.1 Transactions with an invariant property

In this example a transaction is made up of a timestamp, describing when the transaction occurred, and an integer number describing the amount of money transferred in the transaction. The database contains a list of transactions and the overall sum of the amounts transferred in the transactions. The following definitions are written in Clean.

The representation of the type `Timestamp` is irrelevant in this example, hence this type is defined abstract. Two operations are needed on `Timestamp`, namely "`<`" and "`eval`". The type class "`<`" (from the standard library of Clean) denotes an ordering, while the type class "`eval`" (from the standard library of Sparkle) is used to rule out (partially) undefined expressions.

```
:: Timestamp
instance < Timestamp
instance eval Timestamp
```

The operations have to satisfy the following lemma. If the timestamps t_1 and t_2 are not partially undefined, then $t_1 < t_2$ is not partially undefined, either.

$$\forall t_1, t_2 \in \mathit{Timestamp}: \ \mathit{eval}(t_1) \wedge \mathit{eval}(t_2) \Rightarrow \mathit{eval}(t_1 < t_2)$$

Type `Transaction` also has "`<`" and "`eval`" operations, furthermore, two getter operations have been provided as well. Transactions are ordered according to their timestamps, and a transaction is not partially undefined if (and only if) neither of its two components are.

```
:: Transaction = Tx Timestamp Int
timestamp (Tx timestamp amount) = timestamp
amount    (Tx timestamp amount) = amount
instance < Transaction where (<) (Tx t1 a1) (Tx t2 a2) = t1 < t2
instance eval Transaction
    where eval (Tx timestamp amount) = eval timestamp && eval amount
```

The database – given by the synonym type `DB` – is also an instance of the type class "`eval`". (In this simple example program it might be assumed, but it is not obligatory, that the timestamp of the transactions is a primary key.)

```
:: DB :== (Int,[Transaction])
instance eval DB where eval (sum,list) = eval sum && eval list
```

This example is based on the following database operations. The first one creates an empty database, the second one adds a transaction to the beginning of the transaction list, the third one removes the first transaction from the list and, finally, the fourth one sorts the transactions according to their timestamps. These operations describe some basic state transitions of the databases.

```
newDB :: DB
newDB = (0,[])

insertFirst :: Transaction DB -> DB
insertFirst tx=:(Tx tstamp amount) (sum,txs) = (sum+amount,[tx:txs])

removeFirst :: DB -> DB
removeFirst db=:(_,[]) = db
removeFirst (sum,[(Tx tstamp amount):txs]) = (sum-amount,txs)

sortDB (sum,txs) = (sum, isort txs)
```

For sorting we have applied a simple insertion sort function, also used by [20].

```
ins :: a [a] -> [a] | < a
ins e [] = [e]
ins e [x:xs] = if (x<e) [x:ins e xs] [e:x:xs]

isort :: [a] -> [a]  | < a
isort [] = []
isort [x:xs] = ins x (isort xs)
```

Now a simple "scenario" application can be developed, built upon the basic operations, which simulates an interactive session between a database management application and an end-user. The input to this scenario is a database and a transaction. First the transaction is inserted into the database, then the resulting database is sorted, finally the first transaction stored in the (sorted) database is removed.

```
scenario :: DB Transaction -> DB
scenario db tx
   # db = insertFirst tx db
   # db = sortDB db
   # db = removeFirst db
   = db
```

Now this scenario can be rewritten in extended Clean, making use of the object abstraction technique.

```
scenario (.|. obj db) tx
   .#. (.|. obj db) = insertFirst tx (.|. obj db)
   .#. (.|. obj db) = sortDB         (.|. obj db)
   .#. (.|. obj db) = removeFirst    (.|. obj db)
    =  (.|. obj db)
```

As mentioned earlier, the invariant property of databases is that the first component equals to the total sum of the money transferred by the transactions stored in the second component. In Sparkle, the definitions (functions and predicates) required to formulate a theorem should be given in Clean. A function that sums up the amounts of money appearing in a list of transactions will be used here.

```
sumUp :: [Transaction] -> Int
sumUp [] = 0
sumUp [tx:txs] = amount tx + sumUp txs
```

The invariant property can be formalised now:

$$\forall db\ \forall tx:$$
$$(\text{eval obj} \wedge \text{fst obj} = \text{sumUp (snd obj)}) \quad \text{inv} \quad (\text{scenario db tx})$$
$$(\text{eval tx} \wedge \text{eval db} \wedge \text{fst db} = \text{sumUp (snd db)})$$

Let us introduce the following abbreviations:

$$\begin{aligned} \mathrm{I}(x) &= (\text{eval x} \wedge \text{fst x} = \text{sumUp (snd x)}) \\ PRE &= (\text{eval tx} \wedge \mathrm{I}(\text{db})) \end{aligned}$$

The "Invariant" tactic of extended Sparkle will introduce the universally quantified variables *db* and *tx* among the declared variables, and then rewrite the above invariant into the following subgoals.

1. $\forall \text{db}_1$: PRE $\wedge$ db = db_1 $\Rightarrow$ $\mathrm{I}(\text{db}_1)$
2. $\forall \text{db}_1\ \forall \text{db}_2$: PRE $\wedge$ $\mathrm{I}(\text{db}_1)$ $\wedge$ db_2 = insertFirst db_1 tx $\Rightarrow$ $\mathrm{I}(\text{db}_2)$
3. $\forall \text{db}_1\ \forall \text{db}_2$: PRE $\wedge$ $\mathrm{I}(\text{db}_1)$ $\wedge$ db_2 = sortDB db_1 $\Rightarrow$ $\mathrm{I}(\text{db}_2)$
4. $\forall \text{db}_1\ \forall \text{db}_2$: PRE $\wedge$ $\mathrm{I}(\text{db}_1)$ $\wedge$ db_2 = removeFirst db_1 $\Rightarrow$ $\mathrm{I}(\text{db}_2)$

These subgoals can then be proved with Sparkle – the proofs require about 300 steps.

6.2 Dining philosophers with an unless property

This example uses an implementation of Dijkstra's famous "dining philosophers' problem" [6]. In the middle of a dining room there is a table with a big plate of spaghetti. Around the table there are five philosophers spending their lives thinking and eating spaghetti. A philosopher needs two forks for eating spaghetti. However, there are only five forks available, one between each pair of philosophers. Hence two neighbouring philosophers can never eat simultaneously. At the beginning of the program each philosopher is thinking. When a philosopher becomes hungry, he tries to pick up the two forks that he is sharing with his two neighbours. If he

manages to do so, he eats for a while, and then he releases the forks and starts thinking. A hungry philosopher has to wait, if one of the neighbouring philosophers is using the fork shared between them.

The example program uses concurrent ObjectIO processes. It has a graphical user interface for the manipulation of the philosophers. Each philosopher is implemented as a process with its own local state. Moreover, the application also contains a server process that provides the necessary synchronisation. An important part of the server process is the `next_event` function, which controls the critical state transitions of the philosophers.

The local state of the philosopher processes is a value of type `State`.

```
:: State = Thinking | Hungry | Eating
```

The local state of the server process is a list of `States`. The i^{th} element of this list is invariantly equal to the local state of the i^{th} philosopher process.

If a philosopher would like to change its local state from Thinking to Eating, or from Eating to Thinking, the server computes the new states for all of the philosophers. This computation is implemented in the `next_event` function.

```
next_event :: [State] Int -> [State]
```

The function has two arguments. The first one is the local state of the server and the second one is the ordinal number of the philosopher requesting the state transition. The result of the function is the new local state of the server.

In order to present an *unless* property in a very simple context, this concurrent program will be simulated with the following function.

```
process_events:: [State] [Int] -> [State]
process_events states [] = states
process_events states [index:indices]
  | index < 0 || index >= length states        // illegal index
      = process_events states indices          // discarding...
  | otherwise
      # states = next_event states index       // process index
      = process_events states indices          // process rest
```

Since the current implementation of the extended Sparkle system cannot handle rule alternatives, the guards have to be eliminated and the second function alternative has to be reformulated with an `if`-construct. Furthermore, the state transitions have to be made explicit with let-before constructs: this justifies the presence of the second, at first glance unnecessary, such construct in the following function alternative.

```
process_events states [index:indices]
  # states = if (index < 0 || index >= length states)
               states                          // skip illegal index
               (next_event states index)       // process index
  # states = process_events states indices     // process rest
  = states
```

Let us introduce an object that corresponds to the local state of the server process. Then the `process_event` function can be rewritten to extended Clean.

```
process_events :: [State] [Int] -> [State]
process_events (.|. obj1 states1) [] = (.|. obj1 states1)
process_events(.|. obj1 states1) [index:indices]
  .#. (.|. obj1 states2) =
       if ((index < 0) || (index >= length (.|. obj1 states1)))
          (.|. obj1 states1)
          ( next_event (.|. obj1 states1) index )
  .#. (.|. obj1 states3) process_events (.|. obj1 states2) indices
   =  (.|. obj1 states3)
```

The *unless* property expresses the following. Given a Hungry philosopher and his Eating right neighbour, they will not change their states unless the Eating right neighbour starts Thinking. To be less informal, this can be written in the following way.

$\forall i \in domain(states)$:

$$\begin{aligned} &\text{states}_i = \text{Hungry} \ \wedge \ \text{states}_{rightneighbour(i)} = \text{Eating} \\ &\mathbf{unless}(process\_events) \\ &\text{states}_{rightneighbour(i)} = \text{Thinking} \end{aligned}$$

The property "P $\mathbf{unless}_{prog}$ Q" means that during the execution of the program "prog", the statement P remains to hold until the statement Q becomes true. This kind of safety properties can be expressed with the *weakest precondition* operator: for every atomic state transition of the program "prog", the weakest precondition of $P \vee Q$ with respect to the state transition follows from $P \wedge \neg Q$.

$$\forall st \in prog: \quad P \wedge \neg Q \Rightarrow wp(st, P \vee Q)$$

Now let us formulate the *unless* property in a more precise way. To increase readability, the syntax of the extended Sparkle Core is not followed rigorously.

∀states ∀indices ∀i:

```
(eval states) ∧ (eval indices) ∧ (i >= 0) ∧ (i < length states) ∧
(obj1!!i == Hungry) ∧ (obj1!!(rightneighbour obj1 i) == Eating)

UNLESS (process_events states indices)

(obj1!!(rightneighbour obj1 i) == Thinking)
```

The appendix provides the function computing the semantical value of *unless* propositions. Let us investigate what proposition results from applying this semantical function. In the extended Clean version of the function `process_events` there

are two steps, two atomic state transitions affecting the object $\texttt{obj}_1$. The first step is the `if`-construct, and the second step is the recursive call to `process_events`. When reasoning about safety properties, proofs about recursive calls of functions with the same object argument(s) can be omitted. Hence there is only one proposition to prove for the afore-mentioned *unless* property:

```
∀states ∀indices' ∀i ∀states' ∀index ∀indices

(eval states) ∧ (eval indices') ∧ (i >= 0) ∧ (i < length states) ∧
(states!!i == Hungry) ∧
(states!!(rightneighbour states i) == Eating) ∧
¬ (states!!(rightneighbour states i) == Thinking)  ∧
(indices' = [index:indices]) ∧
(states' = if (index < 0 || index >= length states)
              states
              (next_event states index))

⇒

((eval states') ∧ (eval indices') ∧ (i >= 0) ∧ (i < length states') ∧
(states'!!i == Hungry) ∧
(states'!!(rightneighbour states' i) == Eating))
∨ (states'!!(rightneighbour states' i) == Thinking)
```

7 Conclusions and future work

The authors have studied a method that allows the formulation and proof of safety properties (namely invariants and unless) in pure functional languages. The concept of object abstraction has been presented, which is based on contracting functional variables into objects with dynamic (temporal) behaviour. Language constructs describing object abstraction have been introduced into the purely functional programming language Clean. This extended Clean language is considered as an intermediate language: programs written in this language are intended to be generated by an appropriate integrated development environment containing a refactoring tool. Support for the new language constructs will thus be provided by an interactive environment.

Programs written in the extended Clean language are processed by a theorem prover framework. This framework was obtained by enabling Sparkle, the theorem prover designed for Clean, to manage object abstraction and temporal propositions. This paper describes invariant and unless propositions, with a focus on the semantic function computing the meaning of invariant propositions in the logic framework of the Sparkle system.

In the future the object abstraction technique will be generalised to enable the definition of the states of the same object within more than one function.

This will make it possible to express that the atomic state transitions of an object can be spread among many function bodies. This generalisation requires a more sophisticated view of "time" (with respect to temporal operators) and a hierarchical system of state transitions.

The authors also plan to implement an integrated software development environment, which supports object abstraction in the user interface and related refactoring possibilities. This IDE will eliminate the need for programming in extended Clean. Furthermore, the Sparkle framework will be made capable of handling additional temporal propositions, so that it will be possible to express progress propositions (such as "ensures" and "leads-to" [4]) as well. Finally, in order to make temporal reasoning less cumbersome in practice, the authors will provide useful theorems about temporal operators (such as the "weakening" rule or the "conjunction with invariants" rule [13]) as axioms.

The main advantage of the method described in this paper is that in this extended logical framework certain important properties of programs can be expressed conveniently and briefly, at a high level of abstraction. The sophisticated logical operators of temporal logics can neatly express safety and progress properties of programs, and these properties, as the examples of this paper have illustrated, are sensible and useful also in the world of functional programming. Moreover, the addition of theorems about temporal logical operators can make reasoning about programs even less tiring and less complicated.

Another important issue of this approach is that the proofs constructed in the extended Sparkle system are represented in a completely machine processable form. As a consequence, not only the program, but also its proved temporal properties and the proofs themselves can be stored, transmitted and checked by a computer. This allows the transmission of safe code among (components of) applications. A detailed presentation of this proof-carrying code technique can be found in [8].

References

[1] Achten, P., Plasmeijer, R.: Interactive Objects in Clean. *Proceedings of Implementation of Functional Languages, 9th International Workshop, IFL'97* (K. Hammond et al (eds)), St. Andrews, Scotland, UK, September 1997, LNCS 1467, pp. 304–321.

[2] Butterfield, A., Dowse, M., Strong, G.: Proving Make Correct: IO Proofs in Haskell and Clean. *Proceedings of Implementation of Functional Programming Languages*, Madrid, 2002. pp. 330–339.

[3] Butterfield, Andrew: Reasoning about I/O and Exceptions. *Proceedings of Implementation and Application of Functional Languages, IFL'04*, Lbeck, September 8-10, 2004., pp. 33-48.

[4] Chandy, K. M., Misra, J.: *Parallel program design: a foundation.* Addison-Wesley, 1989.

[5] Dam, M., Fredlund, L., Gurov, D.: Toward Parametric Verification of Open Distributed Systems. *Compositionality: The Significant Difference* (H. Langmaack, A. Pnueli, W.-P. De Roever (eds)), Springer-Verlag 1998.

[6] Dijkstra, E. W. *Hierarchical ordering of sequential processes.* Acta Informatica, 1, 115–138, 1971.

[7] Dijkstra, E. W.: *A Discipline of Programming.* Prentice-Hall Inc., Englewood Cliffs (N.Y.), 1976.

[8] Daxkobler K., Horváth Z., Kozsik T.: A Prototype of CPPCC - Safe Functional Mobile Code in Clean. *Proceedings of Implementation of Functional Languages'02*, Madrid, Spain, Sept. 15-19, 2002. pp. 301-310.

[9] Diviánszky P. - Szabó-Nacsa R. - Horváth Z.: A Framework for Refactoring Clean Programs. *6th International Conference on Applied Informatics*, Eger, Hungary January 27-31 2004.

[10] Dowse, M., Butterfield, A., van Eekelen, M., de Mol, M., Plasmeijer, R.: Towards Machine-Verified Proofs for I/O *Proceedings of Implementation and Application of Functional Languages, IFL'04*, Lbeck, September 8-10, 2004., pp. 469-480.

[11] Dowse, M., Butterfield, A.: A Language for Reasoning about Concurrent Functional I/O (Draft) *Proceedings of Implementation and Application of Functional Languages, IFL'04*, Lbeck, September 8-10, 2004., pp. 129-141.

[12] Home of Clean. http://www.cs.kun.nl/~clean/

[13] Horváth Z.: The Formal Specification of a Problem Solved by a Parallel Program—a Relational Model. *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, Tomus XVII. (1998) pp. 173–191.

[14] Horváth Z., Achten, P., Kozsik T., Plasmeijer, R.: Proving the Temporal Properties of the Unique World. *Proceedings of the Sixth Symposium on Programming Languages and Software Tools*, Tallin, Estonia, August 1999. pp. 113–125.

[15] Horváth Z., Achten, P., Kozsik T., Plasmeijer, R.: Verification of the Temporal Properties of Dynamic Clean Processes. *Proceedings of Implementation of Functional Languages, IFL'99*, Lochem, The Netherlands, Sept. 7–10, 1999. pp. 203–218.

[16] Horváth Z., Hernyák Z., Zsók V.: Coordination Language for Distributed Clean. To appear in *Acta Cybernetica*, Szeged, Hungary, 2005.

[17] Horváth Z. - Kozsik T. - Tejfel M.: Proving Invariants of Functional Programs. *Proceedings of Eighth Symposium on Programming Languages and Software Tools*, Kuopio, Finland, June 17-18, 2003., pp. 115-126

[18] Horváth Z. - Kozsik T. - Tejfel M.: Verifying invariants of abstract functional objects - a case study. *6th International Conference on Applied Informatics*, Eger, Hungary January 27-31 2004.

[19] Kozsik T., van Arkel, D., Plasmeijer, R.: Subtyping with Strengthening Type Invariants. *Proceedings of the 12th International Workshop on Implementation of Functional Languages* (M. Mohnen, P. Koopman (eds)), Aachener Informatik-Berichte, Aachen, Germany, September 2000. pp. 315–330.

[20] Kozsik T.: Reasoning with Sparkle: a case study. *Technical Report*, Faculty of Informatics, Eötvös Loránd University, Budapest, Hungary.

[21] de Mol, Maarten. PhD thesis (in preparation), Radboud University Nijmegen.

[22] de Mol, M., van Eekelen, M., Plasmeijer, R.: Theorem Proving for Functional Programmers, Sparkle: A Functional Theorem Prover, Springer Verlag, LNCS 2312, p. 55 ff., 2001.

[23] Peyton Jones, S., Hughes, J., et al. *Report on the Programming Language Haskell 98, A Non-strict, Purely Functional Language*, February 1999.

[24] Plasmeijer, R., van Eekelen, M.: *Concurrent Clean Version 2.0 Language Report*, 2001. http://www.cs.kun.nl/~clean/Manuals/manuals.html

[25] Szabó-Nacsa R., Diviánszky P., Horváth Z.: An Environment for Safe Refactoring Clean Programs. *CSCS 2004, The Fourth Conference of PhD Students in Computer Science*, Szeged, Hungary, July 1-4, 2004.

A The semantics of invariant and unless propositions

The semantics of an invariant and an unless proposition in a program context ψ can be computed according to the following definition:

$$Sem\ ((p\ \texttt{unless}\ f\ cxs\ q), \psi) = \begin{cases} SemFunUnless(p, Def(f, \psi),\ cxs,\ q) & \text{if } cxs = Arity^2(f) \\ undef & \text{otherwise} \end{cases}$$

$$SemFunUnless:\ \mathcal{OP} \times (\langle \mathcal{V}_e^{\bullet} \cup \mathcal{O}^{\mathbf{def}} \rangle \times \langle \mathcal{V}_t^{\bullet} \rangle \times \mathcal{E}^{ext}) \times \langle \mathcal{E}^{ext} \rangle \times \mathcal{P} \to \mathcal{P}$$

$$SemFunUnless\ (p,\ (xs,\ \alpha s,\ e),\ cxs,\ q) = SemExprUnless(p,\ q,\ e,\ True)$$

$$SemExprUnless:\ \mathcal{OP} \times \mathcal{OP} \times \mathcal{E}^{ext} \times \mathcal{P} \to \mathcal{P}$$

$$SemExprUnless(p,\ q,\ (\texttt{let}\ binds\ \texttt{in}\ e),\ pred) = SemExprUnless(p,\ q,\ e,\ pred)$$

$$SemExprUnless(p,\ q,\ (\texttt{step}\ stepbinds\ \texttt{in}\ e),\ pred) = SemExprUnless(p,\ q,\ e,\ (\texttt{binary}\ pred\ \texttt{and}\ (WpImpCalcUn(p,\ q,\ stepbinds))))$$

$$SemExprUnless(p,\ q,\ \_,\ pred) = pred$$

$$WpImpCalcUn:\ \mathcal{OP} \times \mathcal{OP} \times \langle StepBind \rangle \to \mathcal{P}$$

$$WpImpCalcUn(p,\ q,\ stepbinds) = ObjToVarLeft((\texttt{binary}\ p\ \texttt{or}\ q), ObjToVarRight((\texttt{binary}\ p\ \texttt{and}\ (\texttt{unary not}\ q)), \texttt{nil}, stepbinds, \texttt{nil}),\ \texttt{nil}, \texttt{nil})$$

$$Sem\ ((p\ \texttt{inv}\ f\ cxs\ q), \psi) = \begin{cases} SemFunInv(p, Def(f, \psi),\ cxs,\ q) & \text{if } cxs = Arity^2(f) \\ undef & \text{otherwise} \end{cases}$$

$$SemFunInv:\ \mathcal{OP} \times (\langle \mathcal{V}_e^{\bullet} \cup \mathcal{O}^{\mathbf{def}} \rangle \times \langle \mathcal{V}_t^{\bullet} \rangle \times \mathcal{E}^{ext}) \times \langle \mathcal{E}^{ext} \rangle \times \mathcal{P} \to \mathcal{P}$$

$$SemFunInv\ (p,\ (xs,\ \alpha s,\ e),\ cxs,\ q) = SemExprInv(p,\ e,\ (ForallPred(q,\ (\texttt{binary}\ q\ \texttt{implies}\ ObjSubst(Parameter(xs, cxs),\ p)))),\ q)$$

$$SemExprInv:\ \mathcal{OP} \times \mathcal{E}^{ext} \times \mathcal{P} \times \mathcal{P} \to \mathcal{P}$$

$$SemExprInv(p,\ (\texttt{let}\ binds\ \texttt{in}\ e),\ pred,\ q) = SemExprInv(p,\ e,\ pred,\ \texttt{binary}\ q\ \texttt{and}\ BindsToEqs(binds))$$

$SemExprInv(p,$ (**step** *stepbinds* **in** $e)$, *pred*, $q)$
$= SemExprInv(p,\ e,$ (**binary** *pred* **and**
$(ForallPred(q,$ (**binary** q **implies** $WpImpCalc(p,\ stepbinds)))))),\ q)$

$SemExprInv(p,\ \_,\ pred,\ q) = pred$

$WpImpCalc: \mathcal{OP} \times \langle StepBind\rangle \rightarrow \mathcal{P}$

$WpImpCalc(p,\ stepbinds)$
$= ObjToVarLeft(p,\ ObjToVarRight(p, \mathtt{nil}, stepbinds, \mathtt{nil}),\ \mathtt{nil}, \mathtt{nil})$

$ObjToVarLeft: \mathcal{OP} \times (\mathcal{P}, \langle StepBind\rangle, \langle(\mathcal{O}, \mathcal{V}_e^\bullet)\rangle) \times \langle LetBind\rangle \times \langle(\mathcal{O}, \mathcal{V}_e^\bullet)\rangle \rightarrow \mathcal{P}$

$ObjToVarLeft(p,\ (oldp, \mathtt{nil}, changedlist),\ letbinds, leftchangedlist)$
$= CreateForalls(changedlist * leftchangedlist,$
binary *oldp* **implies**
(**binary** $BindsToEqs(letbinds)$
implies $ObjListChangeInPred(changedlist,\ p)$))

$ObjToVarLeft(p,\ (oldp,$ **cons** ((**obj** $o\ x$) **binds** e) $sbs,\ changedlist),$
$letbinds,\ leftchangedlist)$
$= ObjToVarLeft(ObjChange(o,\ ox,\ p),\ (oldp,\ sbs,\ changedlist),$
cons $(ox$ **binds** $e)\ letbinds,$ **cons** $(o, ox)\ leftchangedlist),$
where $ox = NewVar()$

$ObjToVarLeft(p,\ (oldp,$ **cons** (x **binds** e) $sbs,\ changedlist),$
$letbinds,\ leftchangedlist)$
$= ObjToVarLeft(p,\ (oldp,\ sbs,\ changedlist),$
cons $(x$ **binds** $e)\ letbinds, leftchangedlist),$ **if** $x \in \mathcal{V}_e^\bullet$

$CreateForalls: \langle(\mathcal{O}, \mathcal{V}_e^\bullet)\rangle \times \mathcal{P} \rightarrow \mathcal{P}$

$CreateForalls(nil,\ pred) = pred$

$CreateForalls(cons\ (o, x)\ changedlist,\ pred)$
$= CreateForalls(changedlist,$ **forall exprs** $x\ (pred))$

$BindsToEqs: \langle LetBind\rangle \rightarrow \mathcal{P}$
$BindsToEqs(\mathtt{nil}) =$ **constant** *true*

$BindsToEqs($**cons** $(x$ **binds** $e)\ bs)$
$=$ **binary** ((**var** x) **equals** e) **and** $BindsToEqs(bs)$

$ObjToVarRight: \mathcal{OP} \times \langle StepBind\rangle \times \langle StepBind\rangle \times \langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle$
$\rightarrow (\mathcal{P},\ \langle StepBind\rangle,\ \langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle)$

$$
\begin{aligned}
&ObjToVarRight(p,\ changedbinds,\ \texttt{nil},\ changedlist)\\
&\qquad = (newp,\ changedbinds,\ (changedlist\ *\ newchangedlist)),\\
&\qquad\qquad \texttt{where}\ (newp,\ newchangedlist)\ =\ ObjToNewVars_2(p)
\end{aligned}
$$

$$
\begin{aligned}
&ObjToVarRight(p,\ changedbinds,\ cons\ (x\ \texttt{binds}\ e)\ bs,\ oldchangedlist)\\
&\quad = ObjToVarRight(ObjListChangeInPred(changedlist,\ p),\\
&\qquad changedbinds\ *\ (\texttt{cons}\ (x\ \texttt{binds}\ changedexpr)\ \texttt{nil}),\\
&\qquad\qquad ObjListChangeInBinds(changedlist,\ bs),\\
&\qquad\qquad\ (oldchangedlist\ *\ changedlist)),\\
&\qquad\quad \texttt{where}\ (changedlist,\ changedexpr)\ =\ ObjToVarInExpr(e)
\end{aligned}
$$

$$
\begin{aligned}
&ObjListChangeInPred:\ \langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle\ \times\ \mathcal{OP}\ \rightarrow\ \mathcal{OP}\\
&ObjListChangeInPred(\texttt{nil},\ p)\ =\ p
\end{aligned}
$$

$$
\begin{aligned}
&ObjListChangeInPred(\texttt{cons}\ (o,\ x)\ oxs,\ p)\\
&\quad = ObjListChangeInPred(oxs,\ ObjChange(o,\ x,\ p))
\end{aligned}
$$

$$
\begin{aligned}
&ObjListChangeInBinds:\ \langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle\ \times\ \langle StepBind\rangle\ \rightarrow\ \langle StepBind\rangle\\
&ObjListChangeInBinds(oxs,\ \texttt{nil})\ =\ \texttt{nil}
\end{aligned}
$$

$$
\begin{aligned}
&ObjListChangeInBinds(oxs,\ \texttt{cons}\ (x\ \texttt{binds}\ e)\ es)\\
&\quad = \texttt{cons}\ (\ x\ \texttt{binds}\ (ObjListChangeInExpr(oxs,\ e))\ )\\
&\qquad\qquad\qquad ObjListChangeInBinds(oxs,\ es)
\end{aligned}
$$

$$
\begin{aligned}
&ObjToVarInExpr:\ \mathcal{E}^{ext}\ \rightarrow\ (\langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle,\ \mathcal{E}\\
&ObjToVarInExpr(\texttt{var}\ x)\ =\ (\texttt{nil},\ \texttt{var}\ x)
\end{aligned}
$$

$$
\begin{aligned}
&ObjToVarInExpr(\texttt{obj}\ o\ x)\ =\ (cons\ (o,\ ox)\ \texttt{nil},\ \texttt{var}\ ox),\\
&\qquad \texttt{where}\ ox\ =\ NewVar()
\end{aligned}
$$

$$
ObjToVarInExpr(\texttt{basic}\ b)\ =\ (\texttt{nil},\ \texttt{basic}\ b)
$$

$$
\begin{aligned}
&ObjToVarInExpr(\texttt{symbol}\ s\ \sigma s\ es)\\
&\quad = (changedlist,\ \texttt{symbol}\ s\ \sigma s\ changedexprlist),\\
&\qquad \texttt{where}\ (changedlist, changedexprlist)\ =\ ObjToVarInExprList(es)
\end{aligned}
$$

$$
\begin{aligned}
&ObjToVarInExpr(\texttt{apply}\ e_1\ \texttt{to}\ e_2)\\
&\ = (changedlist_1\ *\ changedlist_2,\ \texttt{apply}\ changedexpr_1\ \texttt{to}\ changedexpr_2),\\
&\quad \texttt{where}\ (changedlist_1, changedexpr_1)\ =\ ObjToVarInExpr(e_1),\\
&\qquad (changedlist_2, changedexpr_2)\\
&\qquad\quad = ObjToVarInExpr(ObjListChangeInExpr(changedlist_1,\ e_2))
\end{aligned}
$$

$$
\begin{aligned}
&ObjToVarInExpr(\texttt{case}\ e\ \texttt{of}\ alts)\\
&\quad = (changedlist,\ \texttt{case}\ changedexpr\ \texttt{of}\ alts),\\
&\qquad \texttt{where}\ (changedlist, changedexpr)\ =\ ObjToVarInExpr(e)
\end{aligned}
$$

$$
\begin{aligned}
&ObjToVarInExpr(\texttt{let}\ binds\ \texttt{in}\ e)\\
&\quad = (changedlist, \texttt{let}\ binds\ \texttt{in}\ changedexpr),\\
&\qquad \texttt{where}\ (changedlist, changedexpr)\ =\ ObjToVarInExpr(e)
\end{aligned}
$$

$ObjToVarInExpr(\texttt{step}\ stepbinds\ \texttt{in}\ e)$
$= (changedlist_1 * changedlist_2, \texttt{step}\ changedstepbinds\ \texttt{in}\ changedexpr),$
$\quad \texttt{where}\ (changedlist_1, changedstepbinds) = ObjToVarInBinds(stepbinds),$
$\qquad (changedlist_2, changedexpr)$
$\qquad\quad = ObjToVarInExpr(ObjListChangeInExpr(changedlist_1, e))$

$ObjToVarInExpr(\texttt{let!}\ x_1 = e_1\ \texttt{in}\ e_2)$
$= (changedlist_1 * changedlist_2, \texttt{let!}\ x_1 = changedexpr_1\ \texttt{in}\ changedexpr_2),$
$\quad \texttt{where}\ (changedlist_1, changedexpr_1) = ObjToVarInExpr(e_1),$
$\qquad (changedlist_2, changedexpr_2)$
$\qquad\quad = ObjToVarInExpr(ObjListChangeInExpr(changedlist_1, e_2))$

$ObjToVarInExpr(\bot_\sigma) = (\texttt{nil}, \bot_\sigma)$
$ObjToVarInExprList : \langle \mathcal{E}^{ext} \rangle \to (\langle (\mathcal{O}, \mathcal{V}_e^\bullet) \rangle, \langle \mathcal{E} \rangle$
$ObjToVarInExprList(\texttt{nil}) = (\texttt{nil}, \texttt{nil})$

$ObjToVarInExprList(\texttt{cons}\ e\ es)$
$= (changedlist_1 * changedlist_2, \texttt{cons}\ changedexpr\ changedexprlist),$
$\quad \texttt{where}(changedlist_1, changedexpr) = ObjToVarInExpr(e),$
$\qquad (changedlist_2, changedexprlist)$
$\qquad\quad = ObjToVarInExprList(ObjListChangeInExprList(changedlist_1, es))$

$ObjToVarInBinds : \langle StepBind \rangle \to (\langle (\mathcal{O}, \mathcal{V}_e^\bullet) \rangle, \langle LetBind \rangle)$
$ObjToVarInBinds(\texttt{nil}) = (\texttt{nil}, \texttt{nil})$

$ObjToVarInBinds(\texttt{cons}\ sb\ sbs)$
$= (changedlist_1 * changedlist_2, \texttt{cons}\ changedbind\ changedbinds)$
$\quad \texttt{where}\ (changedlist_1, changedbind) = ObjToVarInBind(sb),$
$\qquad (changedlist_2, changedbinds)$
$\qquad\quad = ObjToVarInBinds(ObjListBindsChange(changedlist_1, sbs))$

$ObjToVarInBind : StepBind \to (\langle (\mathcal{O}, \mathcal{V}_e^\bullet) \rangle, LetBind)$

$ObjToVarInBind((\texttt{obj}\ o\ x)\ \texttt{binds}\ e)$
$= (\texttt{cons}\ (o, ox)\ changedlist, ox\ \texttt{binds}\ changedexpr),$
$\quad \texttt{where}\ ox = NewVar(),$
$\qquad (changedlist, changedexpr)$
$\qquad\quad = ObjToVarInExpr(ObjListChangeInExpr(\texttt{cons}\ (o, ox)\ \texttt{nil}, e))$

$ObjToVarInBind(x\ \texttt{binds}\ e) = ObjToVarInExpr(e), \texttt{if}\ x \in \mathcal{V}_e^\bullet$
$ObjListBindsChange : \langle (\mathcal{O}, \mathcal{V}_e^\bullet) \rangle \times \langle StepBind \rangle \to \langle StepBind \rangle$
$ObjListBindsChange(oxs, \texttt{nil}) = \texttt{nil}$

$ObjListBindsChange(oxs, \texttt{cons}\ sb\ sbs)$
$\quad = \texttt{cons}\ ObjListBindChange(oxs, sb)\ ObjListBindsChange(oxs, sbs)$

$ObjListBindChange : \langle (\mathcal{O}, \mathcal{V}_e^\bullet) \rangle \times StepBind \to StepBind$
$ObjListBindChange(\texttt{nil}, bs) = bs$

$ObjListBindChange(\texttt{cons}\ ox\ oxs, bs)$
$\quad = ObjListBindChange(oxs, ObjBindChange(ox, bs))$

$ObjBindChange : (\mathcal{O}, \mathcal{V}_e^\bullet) \times StepBind \to StepBind$

$ObjBindChange((\texttt{objid}\ z),\ x),\ (\texttt{objid}\ z)\ \texttt{binds}\ e)$
$\quad = x\ \texttt{binds}\ ObjExprChange(e)$

$ObjBindChange((\texttt{objid}\ z),\ x),\ (\texttt{objid}\ y)\ \texttt{binds}\ e)$
$\quad = (\texttt{objid}\ y)\ \texttt{binds}\ ObjExprChange(e),\ \texttt{if}\ \ z \neq y$

$ObjBindChange((\texttt{objid}\ z),\ x),\ w\ \texttt{binds}\ e)$
$\quad = w\ \texttt{binds}\ ObjExprChange(e),\ \texttt{if}\ \ w \in \mathcal{V}_e^\bullet$

$ObjListChangeInExprList: \langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle \times \langle\mathcal{E}^{ext}\rangle \times \langle\mathcal{E}^{ext}\rangle$
$ObjListChangeInExprList(oxs,\ \texttt{nil}) = \texttt{nil}$

$ObjListChangeInExprList(oxs,\ \texttt{cons}\ e\ es)$
$\quad = \texttt{cons}\ ObjListChangeInExpr(oxs,\ e)\ ObjListChangeInExprList(oxs,\ es)$

$ObjListChangeInExpr: \langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle \times \mathcal{E}^{ext} \rightarrow \mathcal{E}^{ext}$
$ObjListChangeInExpr(\texttt{nil},\ e) = e$

$ObjListChangeInExpr(\texttt{cons}(o,\ x)\ oxs,\ e)$
$\quad = ObjListChangeInExpr(oxs,\ ObjExprChange^{ext}(o,\ x,\ e))$

$ObjExprChange^{ext}: \mathcal{O} \times \mathcal{V}_e^\bullet \times \mathcal{E}^{ext} \rightarrow \mathcal{E}^{ext}$
$ObjExprChange^{ext}(o,\ x,\ (\texttt{var}\ y)) = \texttt{var}\ y$
$ObjExprChange^{ext}((\texttt{objid}\ z),\ x,\ (\texttt{obj}\ (\texttt{objid}\ z))\ w) = \texttt{var}\ x$

$ObjExprChange^{ext}((\texttt{objid}\ z),\ x,\ (\texttt{obj}\ (\texttt{objid}\ y))\ w))$
$\quad = \texttt{obj}\ (\texttt{objid}\ z))\ w\ \ \texttt{if}\ y \neq . z$

$ObjExprChange^{ext}(o,\ x,\ (basic\ b)) = basic\ b$

$ObjExprChange^{ext}(o,\ x,\ (\texttt{symbol}\ s\ \sigma s\ es))$
$\quad = \texttt{symbol}\ s\ \sigma s\ ObjExprChangeList^{ext}(o,\ x,\ es)$

$ObjExprChange^{ext}(o,\ x,\ (\texttt{apply}\ e_1\ \texttt{to}\ e_2))$
$\quad = \texttt{apply}\ ObjExprChange^{ext}(o,\ x,\ e_1)\ \texttt{to}\ ObjExprChange^{ext}(o,\ x,\ e_2)$

$ObjExprChange^{ext}(o,\ x,\ (\texttt{case}\ e\ \texttt{of}\ alts))$
$\quad = \texttt{case}\ ObjExprChange^{ext}(o,\ x,\ e)\ \texttt{of}\ alts$

$ObjExprChange^{ext}(o,\ x,\ (\texttt{let}\ binds\ \texttt{in}\ e))$
$\quad = \texttt{let}\ binds\ \texttt{in}\ ObjExprChange^{ext}(o,\ x,\ e)$

$ObjExprChange^{ext}(o,\ x,\ (\texttt{step}\ stepbinds\ \texttt{in}\ e))$
$\quad = \texttt{let}\ ObjListBindChange((\texttt{cons}\ (o,\ x)\ \texttt{nil}), stepbinds)\ binds$
$\qquad\qquad \texttt{in}\ ObjExprChange^{ext}(o,\ x,\ e)$

$ObjExprChange^{ext}(o,\ x,\ (\texttt{let!}\ x_1\ =\ e_1\ \texttt{in}\ e_2))$
$\quad = \texttt{let!}\ x_1\ =\ ObjExprChange^{ext}(o,\ x,\ e_1)\ \texttt{in}\ ObjExprChange^{ext}(o,\ x,\ e_2)$

$ObjExprChange^{ext}(o,\ x,\ (\bot_\sigma)) = \bot_\sigma$
$ObjExprChangeList^{ext}: \mathcal{O} \times \mathcal{V}_e^\bullet \times \langle\mathcal{E}^{ext}\rangle \rightarrow \langle\mathcal{E}^{ext}\rangle$
$ObjExprChangeList^{ext}(o,\ x,\ \texttt{nil}) = \texttt{nil}$

$ObjExprChangeList^{ext}(o,\ x,\ \texttt{cons}\ e\ es)$
$\quad = \texttt{cons}\ ObjExprChange(o,\ x,\ e)\ ObjExprChangeList^{ext}(o,\ x,\ es)$

$ObjSubst:\ \langle EQVars\rangle \times \mathcal{OP} \rightarrow \mathcal{P}$
$ObjSubst(\texttt{nil},\ p) = ObjToNewVars(p)$

$ObjSubst(cons\ ((\texttt{objid}\ z)\ \texttt{equals}\ e)\ eqs,\ p)$
$\quad = ObjPropSubst((\texttt{var}\ ox)\ \texttt{equals}\ e,\ eqs,\ ObjChange((\texttt{objid}\ z),\ ox,\ p)),$
$\qquad\qquad \texttt{where}\ ox = NewVar()$

$ObjSubst(cons\ ((\texttt{var}\ z)\ \texttt{equals}\ e)\ eqs,\ p)$
$\quad = ObjPropSubst((\texttt{var}\ z)\ \texttt{equals}\ e,\ eqs,\ p)$

$ObjPropSubst:\ \mathcal{P}\ times\ \langle EQVars\rangle \times \mathcal{OP} \rightarrow \mathcal{P}$
$ObjPropSubst(prop,\ \texttt{nil},\ p) = \texttt{binary}\ prop\ \texttt{implies}\ ObjToNewVars(p)$

$ObjPropSubst(prop,\ cons\ ((\texttt{objid}\ z)\ \texttt{equals}\ e)\ eqs,\ p)$
$\quad = ObjPropSubst(\texttt{binary}\ prop\ \texttt{and}\ (\texttt{var}\ ox\ \texttt{equals}\ e),\ eqs,$
$\qquad\qquad ObjChange((\texttt{objid}\ z),\ ox,\ p)),\ \texttt{where}\ ox = NewVar()$

$ObjPropSubst(prop,\ cons\ ((\texttt{var}\ z)\ \texttt{equals}\ e)\ eqs,\ p)$
$\quad = ObjPropSubst(\texttt{binary}\ prop\ \texttt{and}\ ((\texttt{var}\ z)\ \texttt{equals}\ e),\ eqs,\ p)$

$ObjChange:\ \mathcal{O} \times \mathcal{V}_e^{\bullet} \times \mathcal{OP} \rightarrow \mathcal{OP}$
$ObjChange(o,\ x,\ \texttt{unary}\ op\ p) = \texttt{unary}\ op\ ObjChange(o,\ x,\ p)$

$ObjChange(o,\ x,\ \texttt{binary}\ p\ op\ q)$
$\quad = \texttt{binary}\ ObjChange(o,\ x,\ p)\ op\ ObjChange(o,\ x,\ q)$

$ObjChange(o,\ x,\ \texttt{quantor}\ q\ p) = \texttt{quantor}\ q\ ObjChange(o,\ x,\ p)$

$ObjChange(o,\ x,\ e_1\ \texttt{equals}\ e_2)$
$\quad = ObjExprChange(o,\ x,\ e_1)\ \texttt{equals}\ ObjExprChange(o,\ x, e_2)$

$ObjChange(o,\ x,\ \texttt{var}\ px) = \texttt{var}\ px$
$ObjChange(o,\ x,\ \texttt{constant}\ c) = \texttt{constant}\ c$

$ObjExprChange^{temp}: \mathcal{O} \times \mathcal{V}_e^{\bullet} \times \mathcal{E}^{temp} \rightarrow \mathcal{E}^{temp}$
$ObjExprChange^{temp}(o,\ x,\ (\texttt{var}\ y)) = \texttt{var}\ y$
$ObjExprChange^{temp}((\texttt{objid}\ z),\ x,\ (\texttt{objid}\ z)) = \texttt{var}\ x$
$ObjExprChange^{temp}((\texttt{objid}\ z),\ x,\ (\texttt{objid}\ y)) = \texttt{objid}\ y\ \texttt{if}\ y \neq z$
$ObjExprChange^{temp}(o,\ x,\ (basic\ b)) = basic\ b$

$ObjExprChange^{temp}(o,\ x,\ (\texttt{symbol}\ s\ \sigma s\ es))$
$\quad = \texttt{symbol}\ s\ \sigma s\ ObjExprChangeList^{temp}(o,\ x,\ es)$

$ObjExprChange^{temp}(o,\ x,\ (\texttt{apply}\ e_1\ \texttt{to}\ e_2))$
$\quad = \texttt{apply}\ ObjExprChange^{temp}(o,\ x,\ e_1)\ \texttt{to}\ ObjExprChange^{temp}(o,\ x,\ e_2)$

$$ObjExprChange^{temp}(o,\ x,\ (\texttt{case}\ e\ \texttt{of}\ alts)) \\ =\ \texttt{case}\ ObjExprChange^{temp}(o,\ x,\ e)\ \texttt{of}\ alts$$

$$ObjExprChange^{temp}(o,\ x,\ (\texttt{let}\ binds\ \texttt{in}\ e)) \\ =\ \texttt{let}\ binds\ \texttt{in}\ ObjExprChange^{temp}(o,\ x,\ e)$$

$$ObjExprChange^{temp}(o,\ x,\ (\texttt{let!}\ x_1\ =\ e_1\ \texttt{in}\ e_2)) \\ =\ \texttt{let!}\ x_1\ =\ ObjExprChange^{temp}(o,\ x,\ e_1) \\ \qquad \texttt{in}\ ObjExprChange^{temp}(o,\ x,\ e_2)$$

$$ObjExprChange^{temp}(o,\ x,\ (\bot_\sigma))\ =\ \bot_\sigma$$

$$ObjExprChangeList^{temp} :\ \mathcal{O}\ \times\ \mathcal{V}_e^\bullet\ \times\ \langle \mathcal{E}^{temp} \rangle\ \rightarrow\ \langle \mathcal{E}^{temp} \rangle \\ ObjExprChangeList^{temp}(o,\ x,\ \texttt{nil})\ =\ \texttt{nil}$$

$$ObjExprChangeList^{temp}(o,\ x,\ \texttt{cons}\ e\ es) \\ =\ \texttt{cons}\ ObjExprChange(o,\ x,\ e)\ ObjExprChangeList^{temp}(o,\ x,\ es)$$

$$EQVars\ =\ \{e_1\ \texttt{equals}\ e_2 \mid e_1 \in \{\texttt{var}\ x \mid x \in \mathcal{V}_e^\bullet\} \cup \mathcal{O},\ e_2 \in \mathcal{E}^{ext}\} \\ Parameter :\ \langle \mathcal{V}_e^\bullet \cup \mathcal{O}^{\texttt{def}} \rangle\ \times\ \langle \mathcal{E}^{ext} \rangle\ \rightarrow\ \langle EQVars \rangle \\ Parameter(\texttt{nil},\ \texttt{nil})\ =\ \texttt{nil}$$

$$Parameter(cons\ (\texttt{obj}\ o\ x)\ xs,\ cons\ e\ es) \\ =\ (cons\ (o\ \texttt{equals}\ e)\ \texttt{nil})\ *\ Parameter(xs,\ es)$$

$$Parameter(cons\ (\texttt{exprvar}\ z)\ xs,\ cons\ e\ es) \\ =\ (cons\ ((\texttt{var}\ (\texttt{exprvar}\ z))\ \texttt{equals}\ e)\ \texttt{nil})\ *\ Parameter(xs,\ es)$$

The definitions above used some simple functions which are not formally defined here. Informally, they calculate the following:

$ObjToNewVars :\ \mathcal{OP}\ \rightarrow\ \mathcal{P}$ Transforms the object identifiers in an object proposition to fresh variables, so it creates an object-less proposition.

$ObjToNewVars_2 :\ \mathcal{OP}\ \rightarrow\ (\mathcal{P}, \langle(\mathcal{O},\ \mathcal{V}_e^\bullet)\rangle)$ Same as the previous, but it returns the applied mapping from objects to variables.

$NewVar :\ \mathcal{V}_e^\bullet$ It gives a fresh variable.

$ForallPred :\ \mathcal{P}\ \times\ \mathcal{P}\ \rightarrow \mathcal{P}$ Here $ForallPred(q, pred)$ creates a "`forall exprs` x,, quantor to $pred$ for every free expression-variable x of q.

CONTENTS

ISSN 0324—721 X

Felelős szerkesztő és kiadó: Csirik János